Querying the Web of Data guaranteeing valid answers with respect to given criteria

Thanh Binh Nguyen

▶ To cite this version:

Thanh Binh Nguyen. Querying the Web of Data guaranteeing valid answers with respect to given criteria. Computer Science [cs]. Université d'Orléans, 2018. English. NNT : . tel-02426935v1

HAL Id: tel-02426935
https://hal.science/tel-02426935v1

Submitted on 2 Jan 2020 (v1), last revised 2 Dec 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE DOCTORALE
MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

Laboratoire d'Informatique Fondamentale d'Orléans

Thèse présentée par :

Thanh Binh NGUYEN

soutenue le : **3 décembre 2018**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline/Spécialité : **Informatique**

L'interrogation du web de données garantissant des réponses valides par rapport à des critères donnés

Thèse dirigée par :

Mirian HALFELD FERRARI ALVES	Professeur, Université d'Orléans
Béatrice MARKHOFF	MCF-HDR, Université de Tours

RAPPORTEURS :

Ladjel BELLATRECHE	Professeur, ISAE - ENSMA
Genoveva VARGAS-SOLAR	CR-HDR, CNRS

JURY :

Ladjel BELLATRECHE	Professeur, LIAS, ISAE-ENSMA
Jacques CHABIN	MCF, LIFO, Université d'Orléans
Mirian HALFELD FERRARI ALVES	Professeur, LIFO, Université d'Orléans
Béatrice MARKHOFF	MCF-HDR, LIFAT, Université de Tours
Pascal MOLLI	Professeur, LS2N, Université de Nantes
Genoveva VARGAS-SOLAR	CR-HDR, LIG Grenoble

Résumé

Le terme Linked Open Data (LOD) (ou données ouvertes liées) a été introduit pour la première fois par Tim Berners-Lee en 2006. L'auteur a proposé un ensemble de principes de connexion et de publication des données sur le Web dans le but de faciliter la récupération et le traitement de ces dernières par les humains et les machines. Depuis, les LOD ont connu une importante évolution. Aujourd'hui, nous pouvons constater les milliers de jeux de données présents sur le Web de données. De ce fait, la communauté de recherche s'est confrontée à un certain nombre de défis concernant la récupération et le traitement de données liées.

Dans cette thèse, nous nous intéressons au problème de la qualité des données extraites de diverses sources du LOD et nous proposons un système d'interrogation contextuelle qui garantit la qualité des réponses par rapport à un contexte spécifié par l'utilisateur. Nous définissons un cadre d'expression de contraintes et proposons deux approches : l'une naïve et l'autre de réécriture, permettant de filtrer dynamiquement les réponses valides obtenues à partir des sources éventuellement non-valides, ceci au moment de la requête et non pas en cerchant à les valider dans les sources des données. L'approche naïve exécute le processus de validation en générant et en évaluant des sous-requêtes pour chaque réponse candidate en fonction de chaque contrainte. Alors que l'approche de réécriture utilise les contraintes comme des règles de réécriture pour reformuler la requête en un ensemble de requêtes auxiliaires, de sorte que les réponses à ces requêtes réécrites ne sont pas seulement les réponses de la requête initiale mais aussi des réponses valides par rapport à toutes les contraintes intégrées. La preuve de la correction et de la complétude de notre système de réécriture est présentée après un travail de formalisation de la notion de réponse valide par rapport à un contexte. Ces deux approches ont été évaluées et ont montré la praticabilité de notre système.

Ceci est notre principale contribution: nous étendons l'ensemble de systèmes de réécriture déjà connus (Chase, Chase & backchase, PerfectRef, Xrewrite, etc.) avec une nouvelle solution efficace pour ce nouveau défi qu'est le filtrage des résultats en fonction d'un contexte utilisateur. Nous généralisons également les conditions de déclenchement de contraintes par rapport aux solutions existantes, en utilisant la notion de one-way MGU.

La preuve de la correction et de la complétude de notre système de réécriture est présentée après un travail de formalisation de la notion de réponse valide par rapport à un contexte.

Abstract

The term Linked Open Data (LOD) is proposed the first time by Tim Berners-Lee since 2006. He suggested principles to connect and publish data on the web so that both humans and machines can effectively retrieve and process them. Since then, LOD has evolved impressively with thousands datasets on the Web of Data, which has raised a number of challenges for the research community to retrieve and to process LOD.

In this thesis, we focus on the problem of quality of retrieved data from various sources of the LOD and we propose a context-driven querying system that guarantees the quality of answers with respect to the quality context defined by users. We define a fragment of constraints and propose two approaches: the naive and the rewriting, which allows us to filter dynamically valid answers at the query time instead of validating them at the data source level. The naive approach performs the validation process by generating and evaluating sub-queries for each candidate answer with respect to each constraint. While the rewriting approach uses constraints as rewriting rules to reformulate query into a set of auxiliary queries such that the answers of rewritten-queries are not only the answers of the query but also valid answers with respect to all integrated constraints. The proof of the correction and completeness of our rewriting system is presented after formalizing the notion of a valid answers with respect to a context. These two approaches have been evaluated and have shown the feasibility of our system.

This is our main contribution: we extend the set of well-known query-rewriting systems (Chase, Chase & backchase, PerfectRef, Xrewrite, etc.) with a new effective solution for the new purpose of filtering query results based on constraints in user context. Moreover, we also enlarge the trigger condition of the constraint compared with other works by using the notion of one-way MGU.

Remerciements

Avec ces quelques lignes, j'aimerais remercier toutes les personnes sans qui ce travail de thèse n'aurait pas pu voir le jour.

Tout d'abord, je tiens à exprimer ma sincère gratitude à mes directrices de thèse, Madame Mirian Halfeld Ferrari et Madame Béatrice Markhoff, pour avoir accepté de diriger cette thèse. Je tiens également à remercier Monsieur Jacques Chabin avec qui j'ai pris un grand plaisir à travailler. Merci pour votre soutien, vos conseils avisés, vos dévouements et encouragements. Vous m'avez beaucoup appris et aidé durant ces années.

Je remercie vivement Monsieur Lajiel Bellatreche et Madame Genoveva Vargas-Solar les rapporteurs de cette thèse. Leurs nombreuses remarques et suggestions ont contribué à rendre ce travail meilleur. Mes remerciements s'adressent également à Monsieur Pascal Molli qui m'a fait l'honneur d'accepter de siéger dans mon jury.

J'aimerais remercier les deux laboratoires: le Laboratoire LIFO de l'Université d'Orléans et le Laboratoire LIFAT de l'Université de Tours, qui m'ont accueilli et m'ont accordé l'indispensable bourse de thèse pour financer ce travail. Je remercie chaleureusement mes collègues qui m'ont aidé, chacun à leur façon, dans la vie académique et quotidienne.

J'adresse mes sincères remerciements à mes amis, ma famille et plus particulièrement mes parents qui m'ont toujours soutenus et encouragés durant toutes ces longues années d'études.

Enfin, je souhaite exprimer ma reconnaissance à deux personnes très chères à mon coeur, sans lesquelles rien de tout cela n'aurait pu être possible: mon épouse et mon petit garçon d'amour. Je vous aime, merci pour vos sacrifices.

Contents

Introduction **1**

I State-of-art **7**

1 Semantic Web Data Querying **9**
1.1 Semantic Web Data (W3C standard) 9
1.1.1 RDF 11
1.1.2 RDFS/OWL 12
1.1.3 Querying RDF data 15
1.1.4 Constraint for Semantic Web Data 17
1.2 LOD querying systems 21
1.2.1 Single SPARQL endpoint 21
1.2.2 Full web querying 22
1.2.3 Federated query system 23
1.3 Querying with ontologies 25
1.3.1 Description Logic 26
1.3.2 Querying data through DL-Lite family 26
1.4 Conclusion 31

2 Rule-Based Query Rewriting **33**
2.1 Rule-based ontology languages 33
2.2 Principles of Semantic Data Integration 40
2.2.1 Data integration principles 40
2.2.2 Ontology-Based Data Integration 41
2.3 Dealing with Inconsistent Databases 43
2.4 Conclusion 45

II Contributions **47**

3 A Constraint-Based Query System **49**
3.1 Querying environment 49
3.2 Background 53
3.3 Graph database, constraints and provenance 55
3.4 Valid query answer: formal definitions 59
3.5 How to obtain valid answers: the naive approach 61
3.6 How to obtain valid answers: the query rewriting approach 61
3.6.1 An Informal Description 63
3.6.2 General schema of the query rewriting process 65
3.6.3 Preprocessing Negative Constraints 66

3.6.4 Algorithm *RewriteQuery* 70
3.6.5 Correctness of Validation through rewriting queries 76
3.6.6 The whole validation process 83

4 Experiments **85**
4.1 General Architecture 85
4.2 The implementation of a prototype on Graal 86
4.3 Different Scenarios of Experiments 88
4.3.1 Comparing the naive and the rewriting approaches 88
4.3.2 Evaluating the use of a context-driven querying system for very large data sets 96

5 Conclusions and Perspectives **101**

Bibliographie **104**

List of Figures

1 The diagram of the Linked Open Data Cloud 2

1.1 Semantic Web stack 10
1.2 An RDF graph with two nodes 11
1.3 An RDF graph 12

2.1 Dependency graphs associated with dependencies from Example 2.4 37

3.1 Query system overview 50
3.2 Module Validator 65

4.1 Architecture of system 86
4.2 Graal system 87
4.3 Comparison of total time of Naive approach and Rewrite approach . 95
4.4 Architecture with MapReduce 97

List of Tables

1.1 DL-Lite axioms 27
1.2 The *goal reduction* of atoms and positive inclusions 30

2.1 Rule-Based query operating algorithms 34

3.1 Set of constraints on $\mathbb{G}$ 51
3.2 Example of local sources 52
3.3 Datasets 69
3.4 Auxiliary Datasets 69
3.5 Intermediate results of Algorithm 7 75
3.6 Intermediate rewritten queries 76

4.1 List of queries and constraints 89
4.2 Evaluation and Verification in the Naive Approach 90
4.3 Evaluation and Verification in the Naive Approach 91
4.4 Queries and Rewritten queries (Rewriting approach) 92
4.5 Rewriting, Evaluation and Verification of Rewriting approach (Dataset 1) 92
4.6 Rewriting, Evaluation and Verification of Rewriting approach (Dataset 5) 93
4.7 Comparison of evaluating time of Naive approach and Rewrite approach 94
4.8 Comparison of total time of Naive approach and Rewrite approach . 94
4.9 Effect of partitioned data (*w.r.t.* condidence factor) on $\mathcal{Q}$'s execution time. 97
4.10 Different steps of query validation and evaluation (time in seconds). 98

Introduction

The last decade has seen a fast evolution of the Semantic Web and impressive growth of Linked Data. The Semantic Web can be presented as an extension of the Web in which data have meaning not only for humans but also for machines. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. All these technologies provide an environment where queries and inferences on data are possible, allowing the existence of a Web of Data with systems capable of supporting interactions over the network. The basis element of this environment is Linked Data which connect related data across the Web by reusing HTTP Internationalized Resource Identifiers (IRIs) and with the use of the *Resource Description Framework* (RDF) to publicly share semi-structured data on the Web. Generally speaking, Web of Data (a.k.a. *Linked Open Data* - LOD) is the whole Linked Data published on the Web which, in common usage, link together according to the principles set out by Tim Berners-Lee in 2006 [27].

According to Linked Open Data Cloud (LOD Cloud) [1] depicted in Figure 1, there are more than 1200 published datasets so far. There exist among them very large Knowledge Bases (KBs), which either are manually crafted as for instance Cyc [2] (a comprehensive ontology and knowledge base about how the world works, which contains about 1.5 million terms) or WordNet [3] (a lexical ontology containing 155327 words for a total of 207016 word-sense pairs), or are automatically constructed [93] as DBpedia [4] (a dataset containing about 3.4 million concepts described by 1 billion triples, including abstracts in 11 different languages, which are extracted data from Wikipedia), Yago [5] (a huge semantic knowledge base containing more than 10 million entities and 120 million facts about these entities harvested from the web [93]) or BabelNet [6] (a multilingual lexicalized semantic network and ontology, containing about 833 million word senses covering 284 languages). They are either domain-general, as the previously mentioned ones, or domain-specific as GeoNames [7] (a dataset providing RDF descriptions of more than 7.5 million geographical features worldwide), MusicBrainz [8] (an open data music database contains about roughly 1.4 million artists, 2 million releases, and 19 million recordings). The largest ones contain millions of entities and billions of facts about them (attribute values and relationships with other entities). Most of those KBs are in the Web of Linked Open Data [27].

[1] https://lod-cloud.net/
[2] http://www.cyc.com/
[3] https://wordnet.princeton.edu/
[4] http://dbpedia.org/
[5] https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/
[6] http://babelnet.org/
[7] http://www.geonames.org/ontology
[8] https://wiki.musicbrainz.org/LinkedBrainz

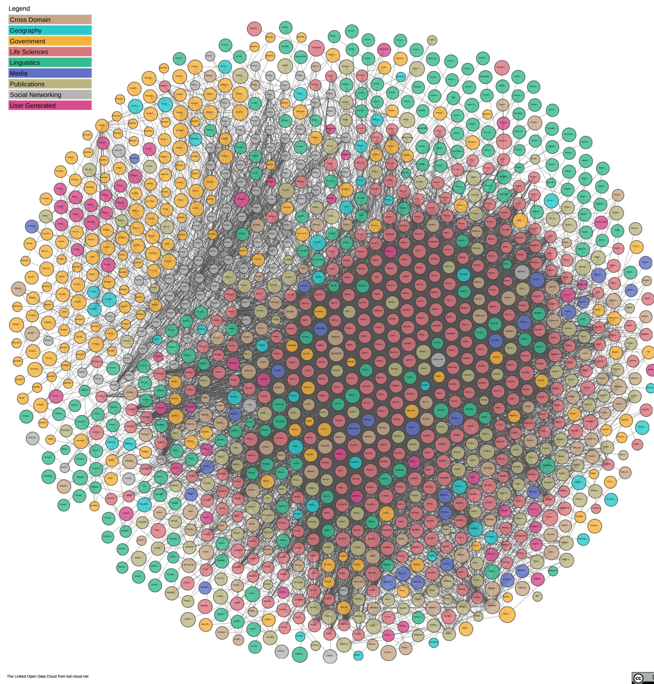

Figure 1 – The diagram of the Linked Open Data Cloud

More and more applications are devised to help humans explore [62] this huge knowledge network, and performing data analysis tasks, and data mining to extract again new knowledge, which may complete or correct the KBs. Very interesting proposals have been recently published concerning this topic [30, 36], which are currently experimented on only one source but can be expected to become even more powerful when they will deal with several linked open data sets. An essential requirement for these tasks is *safe and sound* data retrieved from the Web of Linked Open Data, which is usually done by data collection and preprocessing steps.

For performing data collection and preprocessing, a web application designer may be willing to rely on solutions for accessing several semantic web data source(s) and in that case he will encounter two main approaches: *Query systems* and *Data Integration systems*.

Depending on the purpose and the nature of the data to be retrieved, various LOD query systems have been proposed in literature [71, 47, 6, 84, 77, 87, 87, 59, 10, 37, 92, 70, 85]. They can be classified into 3 categories: one single source, a finite set of query-federated sources and the full web. *Single SPARQL Endpoint* is a centralized database of LOD which allows querying stored data via a web service supporting SPARQL protocol. Basically, it can play as a data warehouse which can extract, transform and load (a.k.a. ETL process) data from (homogeneous or heterogeneous) sources ahead of query time. Then only data in the warehouse are obtained when the system is queried, but not in sources. A *full-web query* system[47] retrieves data based on link-traversal. By this *follow-your-nose* way, it

can explore (probably the entire) Web of Data with priorly unknown sources. A *Federated Query* System, surveyed in [15, 78, 83, 73], proposes a single interface to perform a query on a fixed set of SPARQL Endpoints. The basic difference between this system and a Single SPARQL Endpoint is that it does not have a centralized repository to perform ETL operators. Instead, data remain in data sources and are obtained at query time. The main feature of a Federated Query System is to allow users to query the given SPARQL Endpoints with no a priori knowledge about data sources.

In general, Single SPARQL Endpoint is preferred for scenarios which have to execute very complex queries, whereas the other two systems are suitable for applications requiring the capability of accessing fresh information. Nevertheless, systems such as Federated Query Systems (or Single SPARQL Endpoint harvesting data from multi sources) rarely offer an integrated view of several resources. In other words, the web application designer has to handle the *semantic heterogeneity* issue between data sources, because datasets use generally different schemas even in the same application domain. According to Linked Open Vocabularies (LOV) [9], there exist 650 description of RDFS vocabularies or OWL ontologies defined for and used by datasets in the Linked Data Cloud. To deal manually with this problem, user has to spend time discovering the queried targets by designing lot of queries, until he can actually consider mastering the resulting outputs. Fortunately, the traditional purpose of data integration systems represented in the web by *Ontology-based Data Integration* (ODBI) [91] systems is to automatically manage such issues.

A data integration system essentially operates on semantic mappings between the global-schema/ontology of queries and local-schemas/ontologies of data sources. The semantic-mappings are used to reformulate queries expressed in terms of global-schema/ontologies into sub-queries expressed in terms of local-schemas/ontologies. There exist two basic perspectives to create such mappings and each kind accompanies corresponding algorithms to reformulate the query:

- *Global-as-view* (GAV): the global schema is defined in terms of the source. It uses *query unfolding* algorithm [5] to reformulate queries.

- *Local-as-view* (LAV): source is defined in terms of the global schema. Its well-known query rewriting algorithms are *MiniCon*, *Bucket*, *Inverse-rule* [5].

It should be noted that a data source can be a complete KB with data and complex ontological rules (not limited to traditional integrity constraints). However, both the query systems and the integration systems mentioned above do not handle additional knowledge, which is the knowledge inferred from the facts and rules in the KB. Therefore, it is necessary to have an *Ontological query answering* system (a.k.a. *Ontology-Based Data Access* - ODBA [74]), which computes the set of answers to the query on the facts, while taking implicit knowledge represented in the ontology into account. This process is generally done at local data sources, where local-rules

[9] https://lov.linkeddata.es/dataset/lov

and local-data (*i.e.* KB) are defined. However it does not exclude that this can be performed globally if global-ontological-rules are provided.

In a nutshell, to have sound data in LOD perspective, we need a query system to execute queries and to manage data sources, an integration system to resolve the semantic heterogeneity issue between data sources, an ODBA system to deal with ontological query answering issue, or a combination-system which can play all these roles (for example, an OBDI system can be embedded into the data sources selection module in a Federated system).

It is important to note that sound data retrieved through such a combination-system can be not safe, *i.e.* not valid with respect to constraints. In the context of a classic, centralized database, it is quite obvious that integrity constraints must be checked at the time of modification requests, insert, update, delete, only once, and not at each select query. It is better to check the constraints as little as possible. Clearly, such an approach is not applicable in those scenarios where the data come from different data sources, as in data integration. Moreover, in LOD context, the user is often different from the data creator-owner, so in general he cannot control the data to forbid malicious operators. Even worse in real OBDI scenarios, users can usually only read datasets which are virtually defined through views over autonomous databases.

In addition, the required quality of data, which is defined, for instance, via quality constraints by users, cannot always be identical to those of the data holders. In many cases, user's quality constraints are more general (*i.e.* more expressive) than local ones, which are often integrity constraints. Hence, the data satisfy local constraints at each source, but not necessarily the global constraints of the user who integrates them into a system designed for his needs.

This motivates us to investigate a system on the query side to fulfill user quality requirements. This is a system for validating query answers with respect to a given quality profile, which the user wants to be fulfilled by the results obtained from semantic web data providers, before he can use them in next tasks. In other words, our idea is to extend a query environment over semantic graph databases with a mechanism for filtering consistent answers. The meaning of consistency is in connexion to a user profile that can admit things which are prohibited by others. In our system, a quality profile is a customized context that may be constituted by quality constraints, confidence degrees of data sources or other personalization tools. In this work we only deal with quality constraints and confidence degrees.

User profile is defined via a set of constraints and a set of confident degrees assigned to data sources. Each user's query may have a required confidence degree that is used as a threshold at query time to limit the queried data space to data sources having a sufficient confident degree. For the constraints, we consider a fragment that includes:

- *Positive constraints* which can state that *the presences of some tuples require the presences of some other tuples*. This type corresponds to tuple-generating dependencies that can express the relationships between concepts, roles, attributes and value-domains. For instance, a positive constraint can be used

to state the following statement: "*If a teacher gives a course then there exists a department responsible for this course.*" Moreover, to guarantee the termination of the rewriting query algorithm we propose, constraints are subsumed by the famous weak acyclicity condition defined in [35].

- *Negative constraints* which allow user specifies that *one (or some) instance(s) cannot occur (simultaneously) in data*. In other words, they express disjointness between concepts or properties (so-called denial constraints). *"Nobody can teach and register in Java course in same time"* is an example of this type of constraint.

- *Key constraints* which permit one states that *if some tuples are present then certain components of these tuples must be equal*. Such constraints cover the semantics of functional and identification dependencies in (traditional) databases. For instance, one of key constraints considered in our experiments is "*A person who is the head of a department cannot work for a different department*".

Unlike many studies in literature [22, 42, 81, 80, 13], our goal is neither to modify the queried database nor to try to compute its different possible consistent states with respect to constraints. Instead, our constraints are considered as restrictions imposed on *query results*. Indeed, the constraints are triggered on the basis of atoms in the query's body, and plays as filters in query answers. It is necessary to emphasize that they do not verify data integrity inside the queried sources. Hence, there may be some inconsistencies within data sources, but the answers given to the user are thoroughly filtered to ensure their consistency with respect to user's constraints.

To compute *context-driven answers*, we propose two different methods:

- The *Naive* approach performs the validation process by generating and evaluating sub-queries for each candidate answer with respect to each constraint.

- The *Rewriting* approach uses constraints as rewriting rules to reformulate query into a set of auxiliary queries such that the answers of rewritten-queries are not only the answers of the query but also valid answers with respect to all integrated constraints.

Indeed, the validation process in the second approach consists of two stages: (i) first, reformulate the given query into a set of auxiliary queries by using positive and negative constraints, then evaluate these rewritten queries to obtain candidate answers which at that time are valid *w.r.t.* positive and negative constraints; (ii) those candidate answers continue to be verified with key constraints by using the Naive approach. In this approach, we also propose a preprocessing step that transforms negative constraints into positive ones such that those negative constraints can be treated in the same way as positive ones. Notice that this preprocessing step is performed once for each user profile on a database instance.

Another unique point of our rewriting algorithm is that it can well tackle some situations relating to constants and duplicated variables in constraints. For instance, a case where a positive constraint having a constant which does not appear in the query is still taken into account in our system, but it is ignored in other works.

We provide a complete implementation of the two solutions for the context-driven querying and further present a thorough experimental evaluation thereof, which allows us to evaluate and analyse their practical performance.

Organization

The rest of the dissertation is organized as follows. In the first part of this manuscript, we discuss some related works, trying to positioning our proposal. Contributions are presented in the second part.

Chapter 1 focuses on the Semantic Web Data including data models, ontology, query language and constraint issue for it. This chapter also outlines different kinds of systems for querying Semantic Web Data, as well as a well-known solution for ODBA based on a specific common ontology family: DL-lite.
Chapter 2 extends the issue of querying with ontology in a larger perspective with a more general ontology language: Rule-Based ontology language. Issues related to our contributions are also discussed in this chapter such as query rewriting techniques, ODBA systems, inconsistency in LOD.
In chapter 3, we come to the core part of this thesis, which presents in detail our contributions. It includes syntax and semantics of user quality constraints, the definitions of valid answers in the context of query answering over data sources with confidence.We develop gradually the algorithms that carry out each of the two mentioned approaches for validating query results: the naive based on validating auxiliary sub-queries and the query rewriting based on reformulating query by using constraints.
Chapter 4 presents the results of the experiments we carried out on the algorithms in the previous chapter as well as analysis of them.
Finally, conclusions and further research directions are presented in Chapter 5.

Part I

State-of-art

Chapter 1

Semantic Web Data Querying

The Semantic Web can be presented as an extension of the Web in which data have meaning not only for humans but also for machines. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. All these technologies provide an environment where queries and inferences on data are possible, allowing the existence of a Web of Data with systems capable of supporting interactions over the network. However, to make the Web of Data a reality, relationships among data should be made available. The collection of interrelated datasets on the Web is called Linked Data.

This section offers an overview of structural constraints on the Semantic Web before considering its querying process. An outline of different Linked Open Data (LOD) querying approaches is also presented. The main principles of Description Logic are recalled as well, and we show the ontology querying as an illustration of query-rewriting.

1.1 Semantic Web Data (W3C standard)

The Web, from its inception until the early 2000s, is viewed as a network of inter linked documents. Most of Web pages are usually built by using web languages such as HTML and CSS, and thereby be connected via hyperlinks. In this way, by using Web browsers, users have the possibility of navigating linked documents. Besides Web browsers, search engines are essential tools enabling users to search and traverse such almost infinite information space on the Web.

Although HTML5, CSS3 (newer versions of HTML and CSS) and technologies such as DHTML, AJAX make web pages more responsive to human integration, they are only specifically designed for conveying information in a human understandable way. Well-known techniques implemented in search engines such as page indexing, hit-counts, natural language processing, network analysis can neither adopt the rapid evolution of the Web data nor the users' higher and higher demands. For instance, it is difficult for search engines to recognize similar concepts when they are expressed using different terminologies. Another obstacle is the language dependence, i.e. a search query expressed in one language cannot disclose relevant results in another. The main reason for these limitations is that search technologies are based on keywords matching, but are not designed for answering structured

queries. Contemporary search technologies cannot easily provide integrated information from multiple sources. Users must manually inspect and process the queried results when their information needs are distributed across multiple documents or sources.

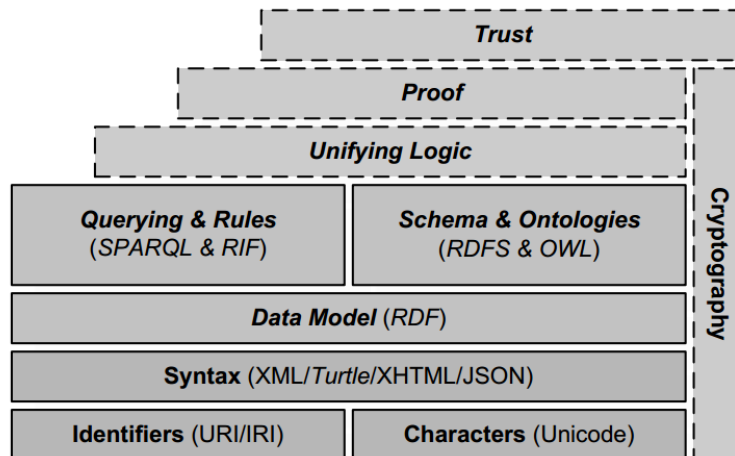

Figure 1.1 – Semantic Web stack

In general, it is not easy for machines to expose what a document is about by scanning its structure. As a consequence, from the beginning of the 2000s, Semantic Web, proposed as an extension of the current web, not only resolves mentioned limitations but also opens a large new research field. The purpose of Semantic Web is to extend the existing web with structure and to provide a mechanism to specify formal semantics of the data on the Web so that information is not just easily accessible for humans, but readily readable and shareable for machines. In this way, the web information can be interpreted, sought, integrated and processed automatically by machines without human intervention. They do so via the use of a data model called Resource Description Framework (RDF) and the use of ontologies, the notion that we will further define in Section 1.3.

In fact, Semantic Web is composed of a stack of technologies that have been designed and recommended by the World Wide Web Consortium (W3C) since 1999. This stack, presented in Figure 1.1, is designed using the philosophy that each layer is built on another layer without modifying it, i.e. the elements described at a layer are compliant with the standards defined at the lower layers. The first two layers of this stack, borrowed directly from existing Web technologies, introduce standards for character encoding and identification, and formal syntax for describing the structure of data. Two middle layers are made up of Semantic Web standards such as RDF for data model, Resource Description Framework Schema (RDFS) and Web Ontology Language (OWL) for expressing the semantics, SPARQL for querying RDF data, etc. Those standards, along with their updates and extensions, have been developed and published by W3C since the last decade. Other higher layers of the stack, in dashed boxes, remain unrealized.

In this section, we concentrate on the two middle layers offering brief introductions to RDF and the two ontology languages RDFS/OWL, before considering how to query the Semantic Web.

1.1.1 RDF

The Resource Description Framework (RDF) is a formal language used as data model for the Web of Data and the Semantic Web. It is a lightweight and flexible way to make statements about things on the web and more generally every abstract or concrete thing. The basic unit of information of RDF is in the form of a triple $\langle s, p, o \rangle$ where each element refers to a subject, a predicate and an object respectively. Such a triple means that predicate p is a relation between s and o. While the subject s refers to the resource that we want to make a statement about, the object o refers either to a resource or to a literal value. A resource or a property in RDF is defined by a Uniform Resource Identifier (URI), a common means providing global identifiers across the Web. The syntax and format of URIs are similar to URLs but a URI does not need to be associated with a Web resource. Naturally, URIs help to solve the problem of identifying common resources among distributed data over multiple sources, a main characteristic of the Semantic Web. Another important advantage of URIs is that they can be dereferenced. That is, one can request a representation of a resource identified by the URI as well as extract the location where potential extra information can be found. The dereferencing ability of URIs plays an important role in the context of Semantic Web because it is not only a simple and effective way to provide global identification but also enables data integration over multiple data sources on the Web.

Definition 1.1 (Triple RDF) *[94] Let U,L and B be three pair-wise disjoint sets of URIs, literals, and blank nodes, respectively. A **triple RDF** is a tuple $\langle s, p, o \rangle$ from $(U \cup B) \times U \times (U \cup L \cup B)$.* □

Interestingly, an RDF triple can be expressed as a directed graph with labeled nodes and arcs. The arcs are directed from the subject to the object. In other words, the arcs represent the predicate relating to the subject (the source node of the edge) and the object (the target node of the edge). The Figure 1.2 illustrates a graph visualizing the triple *(http://example.org/bob, rdf:type, http://example.org/Student)*.

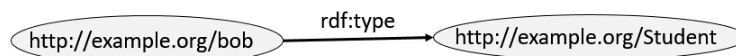

Figure 1.2 – An RDF graph with two nodes

Thus, a set of RDF triple encodes a graph structure whose set of nodes is the set of subjects and objects of triples in the graph.

Definition 1.2 (RDF graph) *[94] An **RDF graph** is a set of RDF triples.* □

Note that nodes in a RDF graph can be IRIs, literals or blank nodes. Figure 1.3 shows a simple example of a graph of four nodes and three edges.

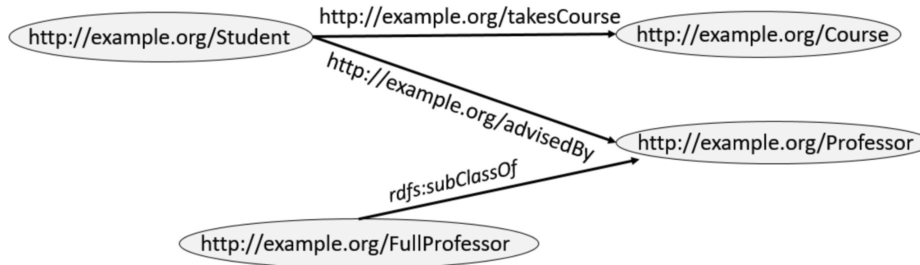

Figure 1.3 – An RDF graph

It is worth noting that a triple RDF $\langle s, p, o \rangle$ can be expressed in the form of an atomic formula in first-order logic (FOL), $P(s, o)$, where P is the name of a predicate, and s and o denote two values in a *binary relation*. In fact, a blank node in an RDF graph is a subject or an object that is identified by neither a URI nor a literal. Thus, in FOL, a blank node can be interpreted as existential variables in the atom. Therefore, in case that s and o are not a blank node in B the set of RDF triples is interpreted as a conjunction of grounded atoms. Otherwise, it is considered as a conjunction of atoms which contains existential variables.

Example 1.1 The following four triples express that Bob is a professor and Bob teaches a course in the informatics department.
Bob :a :Professor
Bob :teaches :_p
:_p :a :Course
:_p :offeredBy "Informatics department"
These triples can be expressed in the following FOL conjunctive atoms:

$$\exists\, p\ [Professor(Bob) \wedge\ teaches(Bob, p) \wedge\ Course(p) \\ \wedge\ offeredBy(p, Informatics\ department)]$$

As discussed above, a binary relation in relational databases can be expressed by using an RDF triple. However, in the Semantic Web, in order to define these binary relations together with the concepts they relate to each other, one uses RDFS or OWL languages that are introduced in more detail in the next subsection.

1.1.2 RDFS/OWL

As shown in the previous section, RDF is a simple data model. Although it can be used to describe quite complex data, its ability to express semantics is very limited. For instance, there is no standard way in RDF to describe a class is a sub class of another one (e.g. *ex:Student is a sub class of ex:Person*) or specify which is the

domain or the range of property. Moreover RDF is domain-independent, i.e. users have to define their own terminology by using vocabularies of particular interest domains. Within the Semantic Web, such shortcomings are overcome by using ontologies. Ontology is a specification of a conceptualization [43]. It aims at describing the concepts in a specific domain and all the relations existing between these concepts. RDFS and OWL are languages proposed by the W3C to define semantic web ontologies. According to the official OWL ontology language description [50], an ontology consists of classes denoting a set of instances, properties denoting binary relationships between classes, and axioms associating specifications of the characteristics with classes and properties. For instance, *Student*, *Course* and *Professor* in the example depicted in Figure 1.3 are concepts in the university domain, and *takesCourse*, *advisedBy* are two properties (a.k.a. predicates) describing relations between them. There are different ontology languages used by the Semantic Web community that mainly differ in their expressiveness. In general, the more expressive an ontology language is, the more complex its computation for reasoning is. Therefore, depending on the purpose and the needs, one has to choose an appropriate ontology formalism that balances expressiveness and computational complexity. In this section, we focus on RDFS and OWL family, two well-known formalism for ontologies that are proposed by the W3C and present in the Semantic Web stack.

RDF Schema, a semantic extension of the RDF, is the least expressive language of the Semantic Web languages for expressing ontologies. It provides a mechanism allowing us to define the vocabulary for RDF statement, define classes or concepts and describe the relationships between them (e.g. subClass, subProperty), specify which properties associate to which kinds of classes, what values they can take and which restriction apply to them. As an extension of RDF, RDFS specifications are expressed in the form of RDF syntax, but uses the namespace `http://www.w3.org/2000/01/rdf-schema#`, commonly associated with the prefix *rdfs*. For example, the statement *ex:FullProfessor rdfs:subClassOf ex:Professor* states that the class *FullProfessor* is a subclass of the class *Professor*, which means that all full professors are professor. By providing universal predicates and properties such as rdfs:Class, rdfs:Resource, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:range, rdfs:domain, RDFS allows us to effortlessly model a particular domain. The following example is the definition of the three concepts and their relations in Figure 1.3 in terms of RDFS predicates.

```
<rdfs:Class rdf:ID="Student"/>
<rdfs:Class rdf:ID="Cours"/>
<rdfs:Class rdf:ID= "Professor"/>
<rdfs:Property rdf:ID="takesCours"/>
    <rdfs:domain rdf:resource = "#Student"/>
    <rdfs:range rdf:resource = "#Cours"/>
</rdfs:Property>
<rdfs:Property rdf:ID="advisedBy"/>
    <rdfs:domain rdf:resource = "#Student"/>
    <rdfs:range rdf:resource = "#Professor"/>
```

```
</rdfs:Property>
```

Although RDFS can provide expressive means, it could not represent more complex knowledge, as well as being far to handle the real world modelling needs. For example, it is impossible to model class disjointness, intersection relationship, property restrictions such as cardinality, value constraints, etc. A common solution to deal with these issues is the use of expressive representation languages based on formal logic, which allows to do logical reasoning on the knowledge and, as a result of that, enables us to reach implicit knowledge. Among such languages, OWL family are widely used and has become an official W3C Recommendation.

The acronym OWL stands for Web Ontology Language. Since 2004 it has been a W3C recommended standard for the modeling of ontologies in 2004, it has been further researched and used popularly in many application domains. OWL is based on Description Logics, and its many different versions correspond to different DLs. However, OWL's syntax is developed on the basis of RDF/RDFS and proposes many new language constructs for describing more complex knowledge, which includes enumerated classes, characteristics of classes/properties, richer typing of properties, cardinality, disjointness of classes, etc. The following example illustrates some new constructs of OWL in RDF/XML syntax.

```
<owl:Class rdf:ID="Person" />
<owl:Class rdf:ID="Student" />
<owl:Class rdf:ID="FullProfessor">
   <rdfs:subClassOf rdf:resource="#Professor"/>
</owl:Class>
<owl:Class rdf:ID="AssociateProfessor">
   <rdfs:subClassOf rdf:resource="#Professor"/>
   <owl:disjointWith rdf:resource="#FullProfessor"/>
</owl:Class>
<owl:Class rdf:ID="Professor">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#advisedBy"/>
         <owl:maxCardinality
            rdf:datatype="&xsd;NonNegativeInteger">1</owl:maxCardinality>
      </owl:Restriction>
   <rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="advisedBy">
   <rdfs:domain rdf:resource="#Student"/>
   <rdfs:range rdf:resource="#Professor"/>
</owl:ObjectProperty>
```

In this example, the class *AssociateProfessor* is defined as a sub class of *Professor* and as being disjoint from another class *FullProfessor*. Besides being specified with

a domain and a range, the property *advisedBy* is also complemented by a cardinality restriction stating that each student is advised by up to one professor.

In 2009, OWL2 was published as a W3C Recommendation, which is essentially a small extension of the original version. It makes OWL more expressive by adding many new languages features such as rule chains, rule composition, type separation, asymmetric rules, reflexive and irreflexive rules, etc. as well as extending some existing constructors such as disjoint classes, qualified cardinality restrictions, data types, so on. Note that in Semantic Web vocabularies, "property" is used instead of "rule", a Description Logic vocabulary.

In OWL2, there are three profiles, each emphasizing different language features: OWL2 EL, OWL2 QL and OWL2 RL. The first one, OWL2 EL, is a fragment that allows polynomial time algorithms for all standard inference types, such as satisfiability checking, classification, and instances checking. It was designed as a language that is particularly suitable for defining ontologies including very large class and rule hierarchies while using only a limited number of OWL features. Differing from the first one, OWL2 QL is designed to enable easier access and query to data stored in databases. It is aimed at applications using very large volumes of instance data, and where query answering is the most important reasoning task. Finally, OWL 2 RL allows standard inference types to be implemented with polynomial time algorithms using rule-based reasoning engines in a relatively straightforward way. It has been designed to allow the easy adoption of OWL by vendors of rule-based inference tools, and it provides some amount of interoperability with knowledge representation languages based on rules.

1.1.3 Querying RDF data

In preceding subsections, we saw that RDF allows us to structure and relate pieces of information, and RDFS and OWL introduce further expressive means for describing complex logical relations. In this subsection, the focus is on querying RDF data. In general, a data model proposal comes together with a language allowing us to query data represented in terms of that model. In case of RDF, that is SPARQL, which is a group of specifications including a query language and protocols that enable us to query and manipulate RDF data.

Different from the approach used in the well-known Structured Query Language (SQL) in relational database that is based on joins between tables, SPARQL uses the matching Basic Graph Pattern based approach to data retrieval. A Basic Graph Pattern (BGP)is an RDF triple in which each of the subject, predicate and object may be a variable. When the variables can be substituted by RDF terms in a subgraph of the RDF data and the result is RDF graph equivalent to the subgraph, we say that the Basic Graph Pattern matches the subgraph of the RDF data.

Although their approaches to retrieve data are different, their syntax is very similar. SPARQL's standard SELECT pattern has the form as

$$SELECT\ \langle result\ template\rangle\ FROM\ \langle data\ set\ definition\rangle$$
$$WHERE\ \langle query\ pattern\rangle.$$

The FROM part is an optional part that indicates the RDF datasets on which the query is performed. While the SELECT part aims at specifying the result we want to exhibit, the WHERE part allows us to restrict the searching graph to the given conditions. The $\langle$*query pattern*$\rangle$ in the WHERE part is a set of triple patterns (a.k.a. Basic Graph Pattern - BGP). Variables in the triple pattens will, on the one hand, store any information resulting from the graph matching problem and, on the other hand, could be combined afterwards in other triples.

Similarly to SQL, SPARQL provides some other keywords with the similar meaning such as ORDER BY, DISTINCT, LIMIT, OFFSET. Besides, it also offers some additional key words such as OPTIONAL, FILTER, and UNION. The OPTIONAL keyword allows us to retrieve data even in the absence of something matching for some triples patterns, i.e. an optional triple is evaluated if it is present and be matched, but the matching does not fail in the case it is not present. Intuitively, OPTIONAL corresponds to OUTER JOIN in SQL. The FILTER allows us to further verify if a variable meets a certain condition. The UNION aims at combining intermediate results provided by sub-queries to produce a result containing any data satisfying at least one of the patterns. Formally, SPARQL is defined as follows:

Let U, B and L be three pair-wise disjoint sets of URIs, literals, and blank nodes, respectively as Definition 1.1, and V be the set of variables.

Definition 1.3 *[45]* A **SPARQL filter condition** is defined recursively as follows:

- If $?x, ?y \in V$ and $c \in (U \cup L)$ then $?x = c$, $?x = ?y$, and $bound(?x)$ are filter conditions;
- If $R1$ and $R2$ are filter conditions then $(\neg R1)$, $(R1 \wedge R2)$, and $(R1 \vee R2)$ are filter conditions.

□

Definition 1.4 *[45]* A **SPARQL expression** is expressed recursively as follows:

- A triple pattern $(U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$ is a SPARQL expression.
- (Optionally) If P is a SPARQL expression, then $P\ FILTER\ R$ is also a SPARQL expression where R is a SPARQL filter condition.
- (Optionally) If $P1$ and $P2$ are SPARQL expressions, then $P1\ AND|OPT|OR\ P2$ are also SPARQL expressions.

□

The above definition is for a *general SPARQL query* whose syntax may have optional clauses. In case it contains a set of triples without UNION, OPTIONAL and FILTER, it is called a *BGP SPARQL query*. Note that a SPARQL query can also be represented as a query graph [73], thus, the semantic of SPARQL query evaluation can be defined as subgraph matching using graph homomorphism.

Definition 1.5 *[45]* Let P be a SPARQL expression and let G be a set of RDF triples. The evaluation of P over G, denoted by $[[P]]_G$, is defined recursively as follows:

- If P is a triple pattern tp, then
  $[[P]]_G = \{\mu | \mu \textit{ is a valuation with } dom(\mu) = vars(tp) \textit{ and } \mu[tp] \in G\}$
- If P is $(P1\ AND\ P2)$, then $[[P]]_G = [[P1]]_G \bowtie [[P2]]_G$.
- If P is $(P1\ UNION\ P2)$, then $[[P]]_G = [[P1]]_G \cup [[P2]]_G$.
- If P is $(P1\ OPT\ P2)$, then $[[P]]_G = [[P1]]_G ⟕ [[P2]]_G$.
- If P is $(P'\ FILTER\ R)$, then $[[P]]_G = \sigma_R([[P']]_G)$.

Each valuation $\mu \in [[P]]_G$ is called a solution for P in G. □

Besides SELECT pattern, SPARQL also offers three other kinds: CONSTRUCT, DESCRIBE, ASK. Unlike a SELECT query that returns a list of variables bound in a query pattern, a CONSTRUCT query returns an RDF graph constructed by substituting variables in the query pattern, while a DESCRIBE query returns an RDF graph describing information about resources that were found. ASK query is a special case for which the returned result is a boolean value indicating whether the query pattern matches.

In 2013, SPARQL 1.1 [88] was introduced as an extension of the original version with a wide range of new features such as property paths, aggregation, subqueries, entailment, federation, etc. In SPARQL 1.0, RDFS/OWL are not taken into account when running queries, by which additional answers can be found through formal entailment mechanisms. This issue is solved in SPARQL 1.1 by the Entailment Regimes mechanism, which offers optional support for such semantics when running SPARQL queries. Another noteworthy feature is *federation* that allows to execute a single query over a selection of SPARQL endpoints by using the SERVICE clause. This feature will be more discussed in Section 1.2.3.

1.1.4 Constraint for Semantic Web Data

In relational databases, the schema consists of rules fixing the vocabulary and the structure (i.e., the relations), plus rules called *constraints*. The term *constraints* is also for RDF data, which denotes vocabulary and structure rules. In the following, we first recall the (relational) database meaning of constraints, noticing that their natural counterparts in Semantic Web are ontological constraints, then we shortly present the principles of proposals of constraints for RDF data.

Constraint is a fundamental part in the database field. Although it has been thoroughly researched on relational databases, it continues to attract the attention of the research community in a new context: constraint in Semantic Web Data.

In the database domain, integrity constraints are restrictions on the contents of the database, and are constructed as a part of the definition of the database itself. They are, in general, a set of descriptions or a set of rules intended to bind some properties of data and to maintain the data integrity during operations on data. Let us recall from [4] that a constraint defining valid set of values for data is called *Domain Integrity Constraint*. With Domain Integrity Constraints, we can define properties on data such as the data type, default value, a specific range value for data, a specific value and so on. These definitions ensure that data will have a right and proper value in the database. Indeed, Domain Integrity Constraint aims only to describe simple properties of an attribute in a relation. For describing more complex properties or relationships between many attributes in the same or different relations, one use data dependencies, commonly called dependencies. A formal form of general dependency can be specified as a first-order logic sentence as follows:

$$\forall x_1, \ldots, x_n \phi(x_1, \ldots, x_n) \rightarrow \exists z_1, \ldots, z_k \psi(y_1, \ldots, y_m)$$

where $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_z\} - \{x_1, \ldots, x_n\}$, ϕ is a (possible empty) conjunction of relation atoms, ψ is a nonempty conjunction of atom or an equality atom of the form $w = w'$, where w and w' are variables appearing in the sentence.

Indeed, there are many types of dependencies in database theory. They can be classified into three fundamental classes as follows[4]:

(i) Full versus embedded: A full dependency is a dependency that has no existential quantifiers.

(ii) Tuple generating versus equality generating: While a tuple-generating dependency (tgd) is a dependency in which no equality atoms occur, an equality-generating dependency (egd) is a dependency for which the right-hand formula is a single equality atom.

(iii) Typed versus untyped: A dependency is typed if there is an assignment of variables to column positions such that (1) variables in relation atoms occur only in their assigned position, and (2) each equality atom involves a pair of variables assigned to the same position.

In Semantic Web Linked Data, axioms in *TBox* of an ontology play the role of representing data semantics, as data dependencies do in relational databases. As OWL is the W3C standard language for modeling ontologies in the Semantic Web, and the logical underpinning of OWL is provided by Description logics, in this section, for simplification, we refer to DLs and OWL to deal with knowledge bases.

Nevertheless, it is important to remark that although the appearance of axioms is quite similar to the constraints of relational databases, they exhibit quite different behavior. In a (relational) database, a constraint belongs to a checking process to *determine whether data respect the constraint restriction*. In a knowledge base,

an axiom is part of an inference process which engenders data from what has been stated.

Let's consider an example on the database of a university in which each professor is required to teach a course. In the ordinary case of relational databases, the following rule is interpreted as an inclusion dependency:

$$\forall X Professor(X) \rightarrow \exists Y TeacherOf(X, Y) \tag{1.1}$$

This inclusion dependency states that for each professor, there must be a course taught by him. For instance, if we insert the fact $Professor(Bob)$ into the database without specifying his courses, the check is performed to determine whether there exists a course associating to Bob in the database. If not, it rises an error and the insertion operator is rejected.
Meanwhile, in the case of Semantic Web, let us consider an axiom (for instance, in the form of DLs) as follows:

$$Professor \sqsubseteq \exists TeacherOf \tag{1.2}$$

Instead of performing a check, an insertion of the instance $Professor(Bob)$ (without his courses) into the knowledge base leads to an inference: Bob teaches some courses, even if we cannot specify exactly what courses they are.

It is worth recalling that relational constraints work under the *Closed-World-Assumption* (CWA). Thus, relational constraints are verified over the instances available in the database. Any missing of information is considered as *false*. For example, consider a database which has one instance $Professor(Bob)$. The inclusion dependency 1.1 is not satisfied , because there is no instance of $TeacherOf$ in the database.

Knowledge bases usually work under the *Open-World-Assumption* (OWA). An incomplete database is described by a set of incomplete extensions of the schema relations and a set of dependencies specifying how the incomplete extensions relate to the actual database instance [69]. The axioms in a knowledge base describe the allowed model, but they do not restrict the allowed model. Thus, the satisfaction of axioms is verified when there exists a model satisfying both *TBox* and *ABox*. This is clearly illustrated in the above example where the axiom $Professor \sqsubseteq \exists TeacherOf$ implies the existence of some (unknown) course taught by *Bob*. Hence, such axioms, in general, cannot be interpreted as relational constraints. In other words, they cannot play the same role as the constraints *validation point of view.*

Indeed, the Semantic Web Linked Data Validation (*i.e.* RDF Validation) has attracted many attentions in last few years. Proposals can be classified in three main categories: SPARQL-based, language-based and inference-based approach.

In SPARQL-based approaches, one attempts to use the SPARQL query language to express the validation constraints. For instance, Knublauch et al. in [52] propose SPARQL Inferencing Notation (SPIN) that uses SPARQL ASK or CONSTRUCT queries to specify constraints. SPIN checks constraints on closed world semantic

and automatically rises inconsistent flags whenever currently available information does not fit the specified constraints. Indeed, SPIN's constraints are embedded in class definitions whereby all instances of the classes are valid with respect to the constraints associated. SPIN allows expressing constraints as SPIN Templates or SPIN Functions that can be reusable and called at runtime. A constraint defined by an SPARQL ASK query checks whether the constraint is satisfied (if the evaluation of the query is false) or violated (if ASK's answer is true), while a CONSTRUCT query returns information about the violation occurred.
The following example illustrates a constraint using ASK which states that credits of each subject must be greater than 1.

Example 1.2
```
ex:Subject
    a           rdfs:Class;
    rdfs:label  "Subject"^^xsd:string ;
    spin:constraint
    [ a             sp:Ask;
      rdfs:comment  "must be at least 1"^^xsd:string ;
      sp:where(
        [   sp:subject    spin:_this
            sp:predicate  ex:credit ;
            sp:object     sp:_credit ;
        ] [ a  sp:Filter ;
            sp:expression [
                a       sp:lt ;
                sp:arg1  sp:_credit ;
                sp:arg2  1
            ]
        ])
    ].
```

RDF validation based on SPARQL has many benefits such as offering a very expressive language that can handle most RDF validation needs and that is supported by most of RDF products. These approaches can perform constraint checking quite effectively. However it is not easy to inspect and understand SPARQL queries for constraint verification, either by human or by machines. This is due to the fact that SPARQL queries may be complex and verbose when representing (even simple) constraints, and one constraint may have more than one way to be expressed. Therefore, SPARQL does not constitute a complete solution to tackle the RDF validation problem [3].

Language-based approaches define a domain specific language to declare the validation rules. OSLC Resource Shape [82], Shape Expression [76], Shapes Constraint Language (SHACL) [54] are typical examples for this type. The common point of these proposals is to define *shapes* that can act as a template specifying the conditions for valid data, so they are also called shapes-based schema languages.

Resource Shape is a high-level RDF vocabulary for describing commonly occurring constraints (but not all constraint) on RDF graphs. These constraints are provided as shapes and other constructs expressed in the form of an RDF graph. Indeed, a shape of RDF graph includes both a description of its expected contents (properties, types) within some operational context (e.g. GET, POST) and the integrity constraints that must be satisfied by the contents in that context [54]. Shape Expression was first proposed in 2014 to provide a human-readable syntax for Resource Shapes.

1.2 LOD querying systems

We know that LOD or RDF is the cornerstone of the Web of Data movement, and SPARQL is W3C recommended query language for RDF. In general, LOD can be stored and processed centralized as most of other data. However, in the majority of cases they are stored across different data sources, and certain queries cannot be answered by retrieving data from only one source. On the other hand, these data sources are owned by different providers, and these owners are not always willing to provide all data schemata or restrict access to raw data. Therefore, depending on the purpose and the nature of the data to be retrieved (*e.g.* storage, data access, etc.), various approaches have been proposed to query LOD. In this section, we classify LOD querying approaches according to the scope of the queried data into 3 categories: one single source, a finite set of query-federated sources and the full web.

1.2.1 Single SPARQL endpoint

LOD querying approaches over one single source are essentially based on a central repository infrastructure that is automatically constructed by harvesting public knowledge-sharing platforms [93]. Such central repository infrastructures are supported by Triple Store Management Systems [1], and provide, in general, an ontology as the schema to design queries. The data managed by triple stores is queried in SPARQL via a SPARQL endpoint, a web service that implements the SPARQL protocol defining the communication processes as well as the accepted and output formats (e.g. RDF/XML, JSON, CSV, etc.). In other words, SPARQL endpoint specified by a URI is a web service that enables users to query linked data via the SPARQL language. According to statistics of LOD cloud [2], 68.14% of the linked data sources offer SPARQL endpoints, while from SPARQL Endpoint Status [1] there are 549 public SPARQL endpoints on the CKAN/DataHub catalog. Examples of such approaches are DBpedia [2], LOD cloud cache [3], Linked Life Data [4] and FactForge [14].

[1] https://www.w3.org/wiki/LargeTripleStores
[2] http://dbpedia.org/sparql
[3] http://lod.openlinksw.com/
[4] http://linkedlifedata.com

The major advantage of central repository infrastructure is the direct availability of locally stored data. This offers benefits such as an excellent query response time and no network communication. However, as the process of collecting and integrating is time-consuming, the data is not always up to date, which may be a serious drawback in the dynamic context of the web. In addition, keeping all data in one place requires not only a lot of storage space but also intensive resources to process large scale data.

1.2.2 Full web querying

Full-Web query systems refer to approaches where the scope of queries is the complete set of Linked Data on the Web [45]. Instead of extracting, transforming, and loading all data from a fixed set of sources before querying it, here all relevant data for a query is discovered during runtime execution. The query evaluation is initialized from a single triple pattern as a starting point and, in an iterative process, relevant data is downloaded by dereferencing URIs which are used to identify Linked Data documents on the web. Parts of the query are iteratively evaluated based on downloaded data, and additional URIs are added, which are dereferenced in the next iteration step. Indeed, an RDF resource may be referred by multiple URIs from multiple independent sources, which may use different ontologies to model their RDF knowledge bases. For instance, URIs can be co-referenced via the *owl:sameAs* property, which is commonly used in Linked Data to state that individuals have the same reference identity. In this way, different sources are connected, and applications can potentially traverse the whole Web of Data by starting from one point. The evaluating process terminates when there are no more URI with potential results to follow. Fully relying on the Linked Data principles and operating through *follow-your-nose* fashion, the only requirement is that the needed data should correctly comply with those principles. This method potentially reaches all data on the web, and the freshest data without requiring any a priori information about relevant data sources as well as the accessing via an SPARQL endpoint provided by sources. But, as the Web of Data is an unbounded and dynamic space, the querying evaluation may not terminate (after a finite number of computation steps or in an acceptable time) if one intends to query the Web of Data on the whole . Thus, in order to implement this approach in practice, it is necessary to restrict the range of queries to a well-defined part of the Web of Data. An example of such a system can be found in [46]. It is worth noting that determining the starting point plays an important role because an inappropriate starting point can increase intermediate results and can significantly influence the completeness of the result. Another drawback of this approach is that URI dereferencing is a time-consuming task, and it cannot handle query patterns having unbound predicates.

1.2.3 Federated query system

In general, a federated system is a collection of cooperating component systems that are autonomous and distributed. A federated-query system refers to a unique interface for querying data from multiple independent given data sources, based on a federation query engine that decomposes the incoming user query into sub-queries, distributes them to data sources, and constructs the final result by combining answers from each source. It is worth noting that the federated-query system being discussed is different from the W3C recommendation of a Federated Query extension for SPARQL 1.1 [88], which supports in executing queries distributed over different SPARQL endpoints by specifying the distant endpoint using the SERVICE keyword. It is quite clear to see that to be able to query by using the SERVICE keyword the query author has to manage all this low-level knowledge. This is the fundamental difference from the approach mentioned above, in which the query system, independently from query's author, will determine automatically how and where a given query is evaluated by analyzing the query itself and the a priori knowledge about data sources. The query processing in a federation framework comprises four phases performed in the following order: query parsing, data source selection, query optimization and query execution. The aim of the first phase of the querying processing, query parsing, is mainly to check the syntax of the input SPARQL queries and to parse them to a set of triple patterns, which are like RDF triples, except that each of the subjects, predicates and objects may be a variable. The results of this phase will be used in selecting sources, as well as optimizing queries in subsequent phases. The main goal of data sources selection phase is to determine the relevant sources containing relevant results against individual subqueries, which in this context are triple patterns or sets of triple patterns, in order to avoid sending them to all data sources. The query optimization phase aims to eliminate unnecessary data transfers between the federated-query system and the sources by (i) using caching, (ii) choosing the appropriate join method, (iii) ordering and grouping the triple patterns. The last phase in query processing is query execution that executes the query following the execution plan specified in the query optimization, and executes the subqueries on relevant sources identified by source selection step.

In some federated system for Linked Open Data, some specific phases can be added to take advantage of some characteristics of Linked Data. For example, in ELITE system [71] a reasoning phase is added between the query parsing and data source selection. The entailment regimes by query rewriting in the reasoner aims to complete query results in terms of a central ontology regardless of the SPARQL endpoint features.

Among the above mentioned phases of the querying processing, data source selection is the most studied phase, as it highly determines the overall performances of Federated Query systems. A single complex query can reach many relations across several domains. On the other hand, there is the fact that a single source can answer for some part of the query, whereas several sources are able to return results for a part of the query. Hence the determination of suitable data sources for each part of

the query is essential in order to increase the efficiency of the querying processing and reduce the overhead of network traffic. SPARQL 1.1 [88], that has become an official W3C Recommendation since 2013, provides the SERVICE keyword to explicitly specify the sources for different parts of the query. However, this approach is rather inappropriate for a flexible environment as Linked Open Data, because it requires query designers to know exactly where the data is. An ideal approach must be transparent in selecting relevant sources (i.e. users can query without a priori knowledge about sources), and be able to adapt to the environment. Many techniques are proposed to deal with these challenges , they are categorized as index-free, index-assisted and hybrid [6, 84, 77, 87].

FedX [87] is an index-free federated engine. In the source selection phase, it sends SPARQL ASK queries to data sources at runtime to specify potential sources. Because of the simplicity of ASK queries, which return boolean values indicating whether a query pattern matches, relevant data sources (which can answer sub-queries corresponding to triple patterns) will be quickly identified. Despite the simplicity, the ASK queries may be too expensive with the growing of the number of triple patterns and the number of data sources. In order to overcome this challenge, cache mechanism can be employed to save the relevance of each triple with each data sources.

In contrast to the index-free fashion, index-assisted approaches build and maintain a source catalog. Such a source catalog provides sources index information which may lead to more efficient and fast source selection, but requires an additional preprocessing stage. In general, a source catalog consists of data mappings which associate RDF terms or complex graph structure with data sources. RDF terms in mappings can be vocabularies in individual datasets, or predicates which allow linking a data entity from one source to other sources (a.k.a. linksets) such as *owl:sameAs*, *rdfs:seeAlso*, *foaf:knows*, etc. On the other hand, it should be noted that common predicates such as *rdf:type*, *rdfs:label*, etc. are used in all data sets. Hence, besides the data mappings, data catalog may also include statistics data which helps to rank data sources. Ranking information, in turn, is useful to reduce the number of relevant sources. Such statistics data can be offered by data sources in terms of their metadata which is usually described by the Vocabulary of Interlinked Datasets (VoID). In case it is not offered by sources, some techniques can be used to calculate the statistics data such as data dump analysis, use crafted queries and aggregate queries or extracting data statistics from query answers. Systems using the catalog technique can be found in DARQ [77], ADERIS [59], Avalanche [10]. The advantage of this strategy is that it can achieve good performance for bound predicates, but they select all available sources for unbound predicates. Furthermore, as using only indexed summaries and possibly out-of-date indexes does not guarantee the completeness of the result set, those systems must deal with index maintenance.

Hybrid systems can be found in [37, 6, 92, 70, 85, 86]. They use both data catalog and ASK queries to optimize the source selection phase. For instance, SPLENDID [37] uses ASK queries to prune inutile pre-selected sources which can-

not return any results for triple patterns having bound subject or object. This significantly reduces the network communication and query processing costs in case a general predicate (e.g. rdfs:label) is used in triple patterns. Another example, ANAPSID [6, 68], uses a heuristic-based source selection in which ASK queries are used to check whether a pre-selected source is relevant for a triple pattern. Furthermore, ANAPSID can update on the fly its data catalog, and offer physical operators who are able to detect the blocked status of sources or bursty network. CostFed [86] is a recently proposed hybrid system that outperforms all the existing ones. It makes use of statistical information collected from endpoints to perform efficient source selection and cost-based query planning.

In the Federated Query approach, queries are answered based on the up-to-date data from original sources. In the web context, this is a major advantage compared to centralized materialized approaches. This is also the case for Ontology-Based Data Integration (OBDI) systems [29, 74]. It is very interesting to notice that, the more Federated Query systems store information in cache or index, about their sources on the one hand, and the user queries on the other hand, the more they resemble traditional integration systems [63]. In particular, indexes storing the relationships between query predicates and source predicates play the same role as GAV or LAV mappings, which will be explained in more detail in Section 2.2 presenting the principles of querying in data integration systems. Compared to the expressive power of OBDI systems, Federated Query systems lack the ability to compile more knowledge into the user query in order to get more complete and more correct results with respect to the user's needs [63]. Nevertheless, the capabilities they developed for automatically harvesting knowledge, about sources and about queries, may be reused to facilitate and enhance OBDI systems.

Note that in these approaches we have studied so far, we are only interested in the topology, organization of the systems and corresponding technologies relating to the usage of query language (*i.e.* SPARQL) as well as the transformation of queries (*e.g.* LAV, GAV), but we have not yet mentioned the querying semantic or how to actually query the RDF data in the presence of ontologies. This issue is going to be considered in detail in the next section.

1.3 Querying with ontologies

Ontologies are the backbone of the Semantic Web [5]. It provides a conceptual view of data and services made available worldwide through the Web [5]. Together with the data model RDF and query language SPARQL, they form the principles of Semantic Web. In this section, we will have a closer look at the ontologies, as well as how to query data with ontologies. To do this, we first recall a family of languages, called Description Logics, which is the formal foundations of the OWL ontology web language recommended by W3C. Then we show that how to deal with new challenges when querying data on the semantic context, that is quite different from that in classical databases. This section is based on the books *Web data management and distribution* [5] and *The Description Logic Handbook* [7]

1.3.1 Description Logic

Description Logics (DLs) is a family of logics that is decidable fragments of first-order predicate logic (FOL). DLs allow expressing and reasoning on complex logical axioms over unary and binary predicates, which is essential to handle ontologies. Indeed, OWL and OWL2 are both based on DLs. Most of the OWL constructs come from DL. The DLs family is composed of many class-based logic languages whose computational complexity depends on the set of constructors allowed in the language. Notice that the class and the property in Semantic Web ontologies are called concept and rule (or role) in DLs, respectively.

A DL knowledge base consists of an intentional part (a.k.a. the *Tbox*) and an assertional part (a.k.a. the *Abox*). The Tbox T defines the vocabulary (terminology) which state inclusions or equivalences between concepts and roles, while the Abox A is a set of facts (assertions) stating memberships of individuals in concepts and role memberships for pairs of individuals. Formally, a TBox consists of statements of the form $B \equiv C$ or $B \sqsubseteq C$ or $R \equiv P$ or $R \sqsubseteq P$, where B, C are class expressions or *atomic concepts* (unary predicates), and R,P are *atomic roles* (binary predicates). An ABox consists of statements of the form $B(a)$ and $R(a, b)$, where B is a class expression, R is a role, and a, b are individuals. Notice that a statement in Tbox may be complex. The following example illustrates that one can build a complex concept by combining atomic concepts and roles with constructors.

$$MasterStudent \equiv Student \sqcap \exists RegisteredTo.MasterProgram$$

In this example, the concepts MasterStudent is defined as a complex concept that is built from two other atomic concepts *Student, MasterProgram*, an atomic role *RegisteredTo* and a conjunction construct. It states that a master student is a student who registered at least one master program. The construct $\exists RegisteredTo.MasterProgram$ is a value restriction, which requires that all the individuals in the relationship $RegisteredTo$, with the concept being described, belong to the concept $MasterProgram$. In general, the equivalence axioms are used to define new concepts from existing concepts, while the inclusion axioms express relations between concepts. The simplest form of inclusion axioms are relations between atomic concepts or roles, which correspond to the *subClassOf* or *subPropertyOf* relations in RDFS.

Although reasoning in all languages in DLs family are decidable, many expressive DLs have high computational complexity in practice. Therefore, we only focus on DLs that is polynomial in reasoning. Among such languages, DL-lite, which is discussed in the following section, provides a good trade-off between expressivity and efficiency.

1.3.2 Querying data through DL-Lite family

As mentioned in the previous section, the DL-Lite family are expressive DLs having a polynomial complexity in reasoning. They have been specially designed for guaranteeing query answering to be polynomial in data complexity. More precisely,

the standard reasoning task in DL-Lite is polynomial in the size of the Tbox, and in LOGSPACE in the size of the Abox [5]. In DL-Lite, we have the negation constructor and unqualified existential restriction on roles and on inverse of roles. In the Tbox of DL-Lite, allowed axioms are concept inclusion statements of the form $B \sqsubseteq C$ or $B \sqsubseteq \neg C$, or existential restriction $\exists R$ or $\exists R^-$, where B, C are atomic concepts, R is an atomic role. It is worth noticing that negation is only allowed in the right hand sides of inclusion statements. Inclusion axioms with negation in the right-hand side are called *negative inclusions* (NI for short), while the inclusion axioms without negation are called *positive inclusions* (PI for short). Besides, DL-Lite also provides functionality axioms (a.k.a Key constraint), which are in the form $(functP)$ or $(functP^-)$ where P is a property and P^- denotes the inverse property of P. The table 1.1 shows some axioms and their corresponding FOL semantics, where B, C are classes, P, Q are properties, and X, Y, Z are variables.

	DL notation	FOL semantics
NI	$B \sqsubseteq \exists\ P$	$B(X) \Rightarrow \exists Y\ P(X,Y)$
	$B \sqsubseteq \exists\ P^-$	$B(X) \Rightarrow \exists Y P(Y,X)$
	$\exists\ Q \sqsubseteq \exists\ P$	$Q(X,Y) \Rightarrow \exists Z P(X,Z)$
	$\exists\ Q \sqsubseteq \exists\ P^-$	$Q(X,Y) \Rightarrow \exists Z P(Z,X)$
	$P \sqsubseteq Q^-$ or $P^- \sqsubseteq Q$	$P(X,Y) \Rightarrow Q(Y,X)$
PI	$B \sqsubseteq \neg C$	$B(X) \Rightarrow \neg C(X)$
	$Q \sqsubseteq \neg P$	$Q(X,Y) \Rightarrow \neg P(X,Y)$
Key	$(func\ P)$	$P(X,Y) \wedge P(X,Z) \Rightarrow Y = Z$
	$(func\ P^-)$	$P(Y,X) \wedge P(Z,X) \Rightarrow Y = Z$

Table 1.1 – DL-Lite axioms

It should be clear that compared with query answering in traditional databases, query answering through ontologies is more complicated because there is the involvement of inferences in query answering which is called *Entailment Regimes mechanism* in SPARQL 1.1 and one has to verify the consistency. In general, to evaluate a query against a dataset D, we need to find valuations of variables in the query such that applying those valuations on all atoms of the body of the query obtains facts "holding". A fact "holds" on dataset D if is a known fact, *i.e.* explicitly exist in the dataset, or if it is a consequence of the facts in the dataset by taking into account rules in the *TBox*, *i.e.* a consequence of logic entailment of the knowledge base. In other words, the first case is a verification concerning data in *ABox* only, while the second one is an inference using both *TBox* and *Abox*.

Thus, depending on the expressiveness of the used ontology the complexity of these inferences are different. For example, in RDFS, all rules are safe [5], i.e. the variables in the head of rule all occur in the body, thus, rules always infer new ground facts. Therefore, answering query through RDFS can be done by applying in a *forward-chaining* manner (a.k.a. *bottom-up* approach) to the initial set of facts until saturation, i.e. until no more facts can be inferred. A simple forward-chaining is depicted in Algorithm 1 [5].

Algorithm 1: Saturation

Input : An ABox A and an RDFS TBox T
Output: A new ABox complemented with inferred facts

```
1  Function Saturation(A, T):
2      F = A;
3      Δ0 = A;
4      repeat
5          Δ1 = ∅;
6          foreach rule σ : condition → conclusion in T do
7              if there exists an homomorphism h such that h(condition) ∈ Δ0
                and h(conclusion) ∉ F then
8                  add h(conlusion) to Δ1 ;
9              F = F ∪ Δ1;
10             Δ0 = Δ1
11     until Δ1 = ∅;
12     return F;
```

In the algorithm, Δ_0 stores facts that need to be considered in each iteration step, while new inferred facts generated at each step are kept in Δ_1. In the first iteration, Δ_0 is all facts in ABox A, but in next steps it is Δ_1 of the previous step, *i.e.* we only consider newly generated facts in the previous iteration. The condition by which a new fact is added into F is that there exists an homomorphism h mapping the condition of rules to the fact, and the application of h on the conclusion of rule does not exist in ABox (including added inferred facts). Algorithm 1 returns a new database, called ontological database, consisting of asserted facts in initial ABox and inferred facts. Then queries are directly evaluated against the ontological database.

However, it is different in case of DL-Lite in which the existential quantifiers are allowed. Indeed, an axiom having existential quantifier in its right-hand-side does not produce new facts from initial facts, but only an incomplete information in the form of atoms that may be partially instantiated. We know that there exists some facts for such atom that is true but we cannot specify exactly the value of existentially quantified variables in the atom. That is the reason why the *backward-chaining* (a.k.a. *top-down*) approach is more appropriate than the *forward-chaining* one in this case.

On the other hand, NIs and Key constraints in DL-Lite can lead to the inconsistency of the knowledge base. That is because NIs express disjointness constraints between classes or properties, and therefore introduce negation in the language, while a Key constraint leads to the inconsistency when we try to equate two distinct constants. The cause of this inconsistency is because DL-Lite adopts the *Unique Name Assumption* (UNA), i.e., different individuals denote different objects. Thus, answering queries through *DL-Lite* ontology needs to address two basic issues: the

inconsistency in the knowledge base and the incompleteness of data.

To deal with the inconsistency problem of the knowledge base, a well-known solution can be found in [5, 55]. In this approach, a *NI-Closure* (Negative Inclusion Closure) is computed from rules in *TBox*, then those rules in NI-Closure are transformed into boolean queries (denoted q_{unsat}). After that, we can evaluate those boolean queries against *ABox* to detect whether there exists one returning *true*, *i.e.* there exists a NI or a key constraint violating with facts in *ABox*. In this case, the knowledge base is *unsatisfiable*.

To overcome the incompleteness of data issues, the *backward-chaining* approach can be used instead of the *forward-chaining* one. Indeed, different from the *forward-chaining* where axioms in the ontology are used as inference rules for deriving new facts from initial fact, in the *backward-chaining* they are used as rewriting rules for reformulating a given query into a set of conjunctive queries (CQs) Q, then these CQs will be evaluated against the data in the *Abox*.

One of the well-know algorithm for reformulating query is the PerfectRef [24] (see Algorithm 2). It takes a conjunctive query q and a TBox T as input, and returns a union of conjunctive queries PR. PR consists of the reformulations of q produced by exploiting the positive inclusions (PIs) in T, which ensures that the answers of query q take into account all the knowledge expressed in T. To do this, for each query q in PR, a new query is generated whenever there is a PI in T applicable to an atom g of q (step (a) of Algorithm 2). PI, in the form $\alpha \Rightarrow \beta$, is applicable to an atom g in a given query q if one of the following conditions is satisfied [24]:

- $g = A(x)$ and A occurs in β.
- $g = P(x_1, x_2)$ and one of the following conditions is satisfied:
  - $\alpha \Rightarrow \beta$ is a role inclusion and P or P^- occur in β.
  - $x_2 = _$ and β is $\exists P$
  - $x_1 = _$ and β is $\exists P^-$

where _ denotes an unbounded existential variable of a query.

The function $gr(g, I)$ (line 9 of Algorithm 2) aims to calculate the *goal reduction* of the atom g using the PI I. Detailed results of the function $gr(g, I)$ is shown in Table 1.2.

Atom g	Positive Inclusion α	$gr(g, \alpha)$
$A(x)$	$A_1 \sqsubseteq A$	$A_1(x)$
$A(x)$	$\exists P \sqsubseteq A$	$P(x, _)$
$A(x)$	$\exists P^- \sqsubseteq A$	$P(_, x)$
$P(x, _)$	$A \sqsubseteq \exists P$	$A(x)$
$P(x, _)$	$\exists P_1 \sqsubseteq \exists P$	$P_1(x, _)$
$P(x, _)$	$\exists P_1^- \sqsubseteq \exists P$	$P_1(_, x)$
$P(_, x)$	$A \sqsubseteq \exists P^-$	$A(x)$
$P(_, x)$	$\exists P_1 \sqsubseteq \exists P^-$	$P_1(x, _)$
$P(_, x)$	$\exists P_1^- \sqsubseteq \exists P^-$	$P_1(_, x)$
$P_1(x_1, x_2)$	$\exists P_1 \sqsubseteq \exists P$ or $\exists P_1^- \sqsubseteq \exists P^-$	$P_1(x_1, x_2)$
$P_1(x_1, x_2)$	$\exists P_1 \sqsubseteq \exists P^-$ or $\exists P_1^- \sqsubseteq \exists P$	$P_1(x_2, x_1)$

Table 1.2 – The *goal reduction* of atoms and positive inclusions

Algorithm 2: PerfectRef

```
   Input  : A conjunctive query q and TBox T
   Output: Union of conjunctive queries PR
 1 Function PerfectRef(q, T):
 2    PR = {q};
 3    repeat
 4       PR' = PR;
 5       foreach q ∈ PR do
 6          (a) foreach g ∈ q do
 7             foreach PI I ∈ q do
 8                if I applicable to g then
 9                   PR = PR ∪ {q[g/gr(g, I)]};

10          (b) foreach g1, g2 ∈ q do
11             if g1 and g2 unify then
12                PR = PR ∪ {τ(reduce(q, g1, g2))};

13    until PR' = PR;
14    return PR;
```

The rewritten queries obtained are then simplified by unifying the pairs of atoms $g1$, $g2$ appearing in each rewritten query q (step (b) of Algorithm 2). More precisely, the function *reduce* replaces the conjunction of two atoms $g1$ and $g2$ with their most general unifier if they can be unified, while the operator τ replaces in the body of a query all existential variables that appear only once (unbound variables) by an anonymous variable. Interestingly, after the simplification step, the bound variables in q can become unbound. Thus, in the next execution of step (a), the PIs that were not applicable to the atoms of the query q in the previous step may become applicable to the atoms in the new query.

It is clear to see that the algorithm *PerfectRef* does reasoning on conjunctive queries and *TBox* but without *ABox*. By separating the reasoning process on the ontology from the data processing, we can gain many practical advantages. One of these is that the reasoning process on ontology depends only on the size of Tbox that is in fact much smaller than Abox. This is important when working on the Semantic Web where the data in Abox is really huge. Another benefit is that we can use an SQL engine for the second step, thereby inheriting well-established query optimization techniques.

Moreover, it is worth noting that reasoning in the presence of PIs, NIs and Key constraints is very complicated, which may lead to an infinite number of non redundant reformulations for some queries [5]. Therefore, we have to control it by some restrictions in the combination of axioms in *Tbox*. For instance, authors in [23] propose *DL-Lite* in which key constraints are forbidden on properties involved in the right hand side of an inclusion axiom. This restriction helps to avoid reformulating a given query into an infinite number of conjunctive queries, each one likely to bring additional answers.

Note that the presented *PerfectRef* is one of typical examples of the *backward-chaining* approach over the DL-family ontology. In the last few years, both Semantic Web and Database communities have proposed many other algorithms that aim at compiling a conjunctive query and an ontology into a union of conjunctive queries. They are not only limited in DL-Lite family but extended over other ontology languages [5, 33, 55, 22]. We will present them in more details in the next chapter, dedicated to rule-based query rewriting in general, thus, in particular, to ontology-based data access.

1.4 Conclusion

Considering the three kinds of LOD querying system previously presented the following remarks can be dressed:

- A Full-Web query system targets its search space in the Web of Data and is appropriate for applications that operate through a *follow-your-nose* fashion. Context-driven querying in this scenario seems unrealistic at the moment due to the unbounded and dynamic nature of Web of Data.

- Single SPARQL Endpoint and Federated systems offer a centralized interface to access (possibly) distributed data and offer a good opportunity for personalizing data access.

The main goal of this thesis is to provide a context-driven querying system on distributed data, particularly on LOD. Single SPARQL Endpoint and Federated systems are therefore our target systems.

Our goal is to provide answers to a user, by integrating data coming from different sources (possibly associated with different confidence degrees) and filtering

them according to user's personal requirements. Both Single SPARQL Endpoint and Federated systems can be enriched by such a tool.

It should also be clear that we are particularly concerned by data validation. However, differently from [5, 55] who deal with the inconsistency problem in the knowledge bases, we are interested in placing the validation step on query answers. More precisely, our proposal is to deal with data sources which can be inconsistent with respect to the constraints fixed by a user context. We do not impose our consistency check on data sources directly, but we propose to filter answers. In this way, we are able to render to the user only data respecting his validity requirements. In this way we ensure the quality of our querying system.

This viewpoint differs from the work in [5], where before executing queries against data on ABox, one verifies the inconsistency of the ABox with respect to TBox. Their approach is composed by different steps including computing *NI-Closures*, transforming them into boolean queries and evaluating these queries against ABox (in oder to determine whether facts in ABox violate NI or key constraints in TBox). As it will be presented in Chapter 3, our solution includes a preprocessing step which may recall the transformation of *NI-Closures* in [5]. However, the goal of our preprocessing step is to create auxiliary datasets which are going to be used as support for query rewriting.

Finally, our constraints are seen in a database perspective [25] (and not in an ontological perspective where they are considered as inference rules). Due to this aspect, the way handling constraints in our solution is different from the one in approaches like [5, 55]. This is presented in detail in the next chapter.

Chapter 2

Rule-Based Query Rewriting

There exist many proposals focusing on rule-based query-rewriting in the database literature, that have been developed to address a very wide range of issues, from query optimization to data integration, through inconsistent databases repairing or ontology-based data access. To deal with those issues, highly famous algorithms have been devised, and regularly improved: Saturation [5], PerfectRef [24], Chase [35, 31, 64, 18], Chase&Backchase [33], Unfolding [5], MiniCon [75], Repair computation algorithms [55, 22], XRewrite [40], etc. Even though we did not find authors who clearly analyze the relationships between all of them, we noticed that they all rely on the fundamental idea of *applying rules* $\phi \rightarrow \psi$, either from *left to right* (forward chaining) or from *right to left* (backward chaining). The considered rules are either ontological constraints, view definitions, GAV or LAV mappings, or data exchange rules, etc. We summarize this observation in Table 2.1, where the notation asterisk indicates that there are restrictions in the syntax of constraints to ensure the termination of the algorithm.

At the end of Chapter 1, we presented the Semantic Web ontology querying in the case where the ontology languages are based on different Description Logics, a family of logics that has been designed for knowledge representation and reasoning. We also studied a concrete case of query-rewriting technique, *PerfectRef*. In this chapter, the Semantic Web ontology querying is considered in a wider perspective with *rule-based ontology languages*. Particularly, we will consider in detail the *chase procedure* and its terminating conditions. We also recall the principles of OBDI, that can play a role for controlling query results, and we think it could be enriched with Federated Query Systems techniques, as noticed in Section 1.2.3. This chapter ends with an overview of the inconsistent database problem, one of the issues with which our contribution deals.

2.1 Rule-based ontology languages

We know ontologies are modeled by using formal languages called ontology languages. Those languages, in general, use First-Order Logic (FOL) (a.k.a. predicate logic) as the formal foundation. We also know that Description Logics is a well-known family of knowledge representation languages in modeling ontologies. DLs are widely used in semantic web community because they are decidable fragments of FOL and allow expressing and reasoning on complex axioms over unary and binary

Algorithm	**Applying rule**	**Type of rules**	**Purpose**
Saturation [5]	forward	ontological constraints (TGDs, EGDs)	generate new facts
Chase [35]	forward	ontological constraints (TGDs, EGDs)	generate new facts
Tableau [4]	forward	database constraints (TGDs, EGDs)	generate new queries & enrich answer set
PerfectRef [24]	backward	ontological constraints (TGDs)	generate new queries & enrich answer set
XRewrite [40]	backward	ontological constraints (TGDs*)	generate new queries & enrich answer set
C&BC [33]	backward	data exchange rules	data exchange
Ours[25]	backward	database constraints (TGDs*, NC)	Filter answer set

Table 2.1 – Rule-Based query operating algorithms

predicates.
Another ontology language family which attracts a lot of attention from the Knowledge Representation and database communities in the last few years is *rule-based ontology languages* [19, 8]. In such languages, tuple-generating dependencies (TGDs) are used for ontological modeling and reasoning, so they are also known as TGD-based ontology languages. Indeed, a TGD is a datalog rule extended by allowing existential quantified variables in its head as the form $\forall \mathbf{X},\mathbf{Y}\ \phi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\ \mathbf{Z}\ \psi(\mathbf{X},\mathbf{Z})$where ϕ and ψ are conjunctions of atoms over a relations schema. Intuitively, different from DL-based ontology languages which support only unary and binary relations, rule-based ontology languages allow expressing and reasoning over arbitrary arity relations.

Example 2.1 The following rules are TGDs describing constraints on a university database

$$\sigma_1 : professor(X_{id}) \rightarrow \exists X_{course}\ teacherOf(X_{id}, X_{course})$$
$$\sigma_2 : teacherOf(X_{id}, X_{course}) \rightarrow \exists X_{dep}, X_{org}\ worksFor(X_{id}, X_{dep}, X_{org})$$

The universal quantifiers in TGDs are usually omitted. □

Recall that to deal with the query answering over DL-Lite in a forward-chaining manner, we build an ontological database by using rules in *TBox* to saturate the given database (Algorithm 1), then queries are evaluated directly against the new database. This is similar in the case of TGD-based ontologies where one computes models, called *canonical model* (a.k.a. universal model), then we can evaluate queries over these models. Such model can be constructed by using *chase procedure*, a well-known technique developed by the database community for reasoning with constraints [60, 61, 90, 79, 4].
Roughly speaking, the chase procedure adds new atoms to the extensional database

to satisfy TGDs by applying a sequence of steps, where each step enforces a constraint that is not satisfied by the current instance. Note that added tuples (instantiated atoms) involve possibly null values which act as a witness for the existentially quantified variables in the atoms.

TGD Chase Step [72]Let D be a database for a relational schema $\mathcal{R}$ and σ be the TGD on $\mathcal{R}$ of the form $\forall \mathbf{X},\mathbf{Y}\ \phi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\ \mathbf{Z}\ \psi(\mathbf{X},\mathbf{Z})$. If there exists a homomorphism h that maps the atoms of $\phi(\mathbf{X},\mathbf{Y})$ to atoms of D (*i.e.* $h(\phi(\mathbf{X},\mathbf{Y})) \subseteq D$), we say that (σ, h) is a *trigger* for D. If there is no extension h' of h such that $h'(\psi(\mathbf{X},\mathbf{Z})) \subseteq D$ then (σ, h) is said to be an *active trigger* for D. h' is extended from h by assigning new fresh nulls to the existential variables in ψ. We say that the TGD σ is applicable to D if (σ, h) is a *trigger* (active or not) for D, *i.e.* the homomorphism h exists. □

In the case (σ, h) is an active trigger, the chase step is *standard-chase*. The following algorithm describes the standard chase procedure for a database D and a set of TGDs Σ, which consists of an exhaustive application of the standard TGD chase steps.

Algorithm 3: Standard TGD chase

Input : A database D and a set of TGDs Σ
Output: A universal model of D and Σ

```
1 Function StandardTGDChase(D, Σ):
2     D' = D;
3     while exists an active trigger (σ, h) for D' do
4         Let h' be the extension of h;
5         D' = D' ∪ h'(head(σ))
6     return D';
```

The final result of Algorithm 3 is a new database, denoted $chase(D, \Sigma)$, whose instances satisfy all constraints (TGD) in Σ.

Example 2.2 Consider an university database
$D = \{professor(Bob), professor(Tom), teacherOf(Alice, Java)\}$
and the set of TGDs Σ of Example 2.1. First, we apply σ_1 to $professor(Bob)$ (resp. $professor(Tom)$), which adds $teacherOf(Bob, N_1)$ (resp. $teacherOf(Tom, N_2)$) into D', and σ_2 to $teacherOf(Alice, Java)$, which adds more $worksFor(Alice, N_3, N_4)$ into D'. Then, in the next iteration, σ_2 is applied on $teacherOf(Bob, N_1)$ (resp. $teacherOf(Tom, N_2)$) to add $worksFor(Bob, N_5, N_6)$ (resp. $worksFor(Tom, N_7, N_8)$) into D'. Thus, we yield a finite chase $chase(D, \Sigma)$ as follows:
$chase(D, \Sigma) = D \cup \{teacherOf(Bob, N_1), teacherOf(Tom, N_2),$
$worksFor(Alice, N_3, N_4), worksFor(Bob, N_5, N_6), worksFor(Tom, N_7, N_8)\}$ □

It is easy to see that the standard chase algorithm is very similar to Saturation (algorithm 1) in DL-Lite case. The condition for a rule or a TGD constraint to

generate a new fact is that exists a homomorphism h and its extension h' such that $h(body(rule))$ is a fact in the current instance while $h'(head(rule))$ is not.The chase procedure in the standard TGD chase is *restricted*, because only active triggers are considered at each chase step. However, checking whether a trigger is active can be difficult in practice. So if all triggers (not only the active ones) are considered at each iteration, the chase procedure is known as *oblivious*. It is the simplest variation of the standard chase (a.k.a. the naive chase), but it can add atoms to the database even if it is not necessary, because there is no checking whether the TGD considered is already satisfied. Furthermore, TGDs are chosen by a nondeterministic way at each step, and the result of chase step depends on the presence of other atoms constructed by previous steps, so standard chase and oblivious one are nondeterministic and difficult to analyze from a theoretical point of view. Thus, to deal with the two above issues, several chase variants can be found in literature such as *semi-oblivious chase* [64], *unrestricted Skolem chase* [64, 89, 65], *core chase* [31], *frugal chase* [53], etc.

Algorithm 3 stops when there are no other active trigger to be applied. Nevertheless, in many cases the chase procedure does not terminate. The following example shows one of these cases.

Example 2.3 Given an instance $D = \{R(a,b)\}$ and a set of TGDs $\Sigma = \{\sigma_1 : R(x,y) \rightarrow \exists z R(x,z)\}$. Clearly as z is an existential variable so there is no active trigger in Σ for any instance. When applying oblivious chase, we obtain an infinite chase sequence which adds infinitely atoms into the instance $D = \{R(a,b), R(a,N_1), R(a,N_2), R(a,N_3), \ldots\}$, where N_i are new fresh null values.

For a given instance and a given set of dependencies Σ, checking if Σ has terminating chase is undecidable [31]. This motivated the research community to find classes of TGDs that ensure the termination of chase procedure. Terminating chase conditions can be found in the literature [35, 49, 66, 31, 39, 64, 19, 16]. Among these approaches, *weak acyclicity* [35, 34] is a well-known sufficient condition on a set of dependencies for the termination of the chase. Roughly speaking, weak acyclicity checks if the set of TGDs does not have a cyclic condition such that a new null value forces the adding of another new null value [72].

Definition 2.1 (Weakly acyclic) *[35, 34]*
A *position* is a pair (R,i) (which we write R^i) where R is a relation symbol of arity r and i satisfies $1 \leq i \leq r$.
The *dependency graph* of a set Σ of TGDs is a *directed edge-labeled graph* $G_\Sigma = (V,E)$ where the set of vertexes V represents the *positions* of the relation symbols in Σ; and, for every TGD σ of the form $\forall \mathbf{X},\mathbf{Y}\ \phi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\ \mathbf{Z}\ \psi(\mathbf{X},\mathbf{Z})$ there is an edge $(R^i, S^j) \in E$ if one of the following holds:

- some $x \in \mathbf{X}$ occurs in R^i in ϕ and in S^j in ψ (in this case the edge is labeled as *universal*)

- some $x \in \mathbf{X}$ appears in R^i in ϕ and some $z \in \mathbf{Z}$ occurs in S^j in ψ (in this case the edge is labeled as *existential*).

Σ is *weakly acyclic* if its dependency graph has no cycles with an existential edge. □

Example 2.4 *[72]* Consider database schema $\mathcal{R} = \{S, R\}$, with $arity(S) = 1$ and $arity(R) = 2$. The set of positions in R is $\{(S, 1), (R, 1), (R, 2)\}$.
Let Σ_1 be a set of TGDs containing the following dependency over $\mathcal{R}$:

$$\sigma_{11} : S(x) \to \exists y R(x, y)$$

let Σ_2 contain the following dependencies:

$$\sigma_{21} : S(x) \to \exists y R(x, y)$$

$$\sigma_{22} : R(x, y) \to \exists z R(x, z)$$

and, finally, let Σ_3 be a slight modification of Σ_2:

$$\sigma_{31} : S(x) \to \exists y R(x, y)$$

$$\sigma_{32} : R(x, y) \to \exists z R(y, z).$$

Figure 2.1 illustrates the dependency graphs associated with Σ_1, Σ_2 and Σ_3 in which the existential edges are represented as dotted lines. It is easy to see that Σ_1 is weakly acyclic as the dependency graph does not contain any cycles; Σ_2 is weakly acyclic as its dependency graph has a cycle going only through universal edges; while Σ_3 is not weakly acyclic as it has a cycle going through an existential edge. □

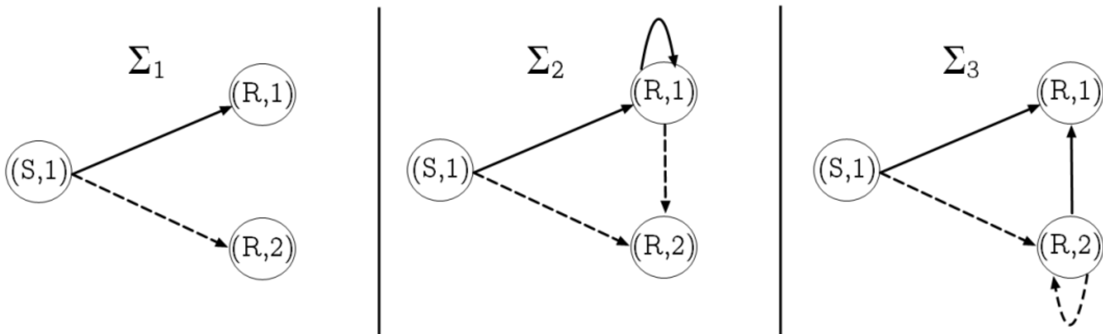

Figure 2.1 – Dependency graphs associated with dependencies from Example 2.4

We say that a set Σ of TGDs and EGDs is weakly acyclic if the set $\Sigma' \subseteq \Sigma$ consisting of the TGDs in Σ is weakly acyclic. Note that given a set Σ of TGDs, testing if Σ is weakly acyclic is polynomial in size of Σ [72].

Theorem 2.1 *[35, 34]* For every weakly-acyclic set Σ of TGDs and EGDs, there are integers b and c upper bounded by the size of Σ such that for every instance A,

1. every chase sequence of A with Σ terminates, and
2. A^{Σ} can be computed in $O(|A|^b)$ steps and in time $O(|A|^c)$.

□

It is well-known that the chase of D with respect to Σ is a *universal model* of D with respect to Σ, *i.e.* there exists a homomorphism from $chase(D, \Sigma)$ onto all models of D with respect to Σ, denoted as $mods(D, \Sigma)$ [35]. This result implies that the answer to a conjunctive query q with respect to a database D and a set of TGDs Σ can be evaluated on the chase for D and Σ, *i.e.* $ans(q, D, \Sigma) = q(chase(D, \Sigma))$. In other words, the chase can be considered as a formal algorithmic tool for query answering under TGDs. Terminating chase conditions, for instance weakly acyclic, guarantees the decidability of query answering under TGDs, which in general is undecidable under arbitrary TGDs [11] even when the schema and the set of TGDs are fixed [18], or even when the set of TGDs is a singleton[8].

Nevertheless, decidability of query answering is not enough when dealing with very large databases. In such case, query rewriting (backward-chaining) is the favorite approach, and first-order rewritability [24] is a desirable property. More precisely, given a conjunctive query q, a database D and a set of TGDs Σ, a *perfect rewriting* q_{Σ} is built from q by taking into account the semantic consequences of the TGDs. Then, the answer to q with respect to D and Σ is obtained by evaluating the q_{Σ} directly over D, *i.e.* $ans(q, D, \Sigma) = q_{\Sigma}(D)$. Note that a conjunctive query guaranteeing the first-order rewritability can be equivalently rewritten in (non-recursive) SQL, which allows to exploit all the optimization capabilities of the underlying RDBMS.

In DLs world, the members of the DL-Lite family guarantee the first-order rewritability of conjunctive query answering. Meanwhile in TGD-based ontology, several classes of TGDs ensuring first-order rewritability have been investigated in the last few years such as Linear TGDs [20], sticky TGDs [21, 38].

A TGD is linear iff it contains only a singleton body atom. Despite its simplicity, linear TGDs are powerful enough to express "conditional inclusion dependencies". In fact, inclusion dependencies can be equivalently written as TGDs with just one body-atom and one head-atom without repeated variables and with constant values in the body. Moreover, linear TGDs are strictly more expressive than DL-Lite R, which forms the OWL 2 QL profile of W3Cs standard ontology language for modeling Semantic Web ontologies [39].
A set of TGDs is sticky if it allows joins to appear in body rule which are not expressible via linear TGDs. The key idea underlying stickiness is to ensure that, during the chase, terms which are associated with body-variables that appear more than once (i.e., join variables) are always propagated (or “stick”) to the inferred atoms [39].

Gottlob et al in [39] proposed a backward-chaining rewriting algorithm which constructs a union of conjunctive queries from a conjunctive query and a set of linear and sticky TGDs. Particularly, a conjunctive query is rewritten by exhaustively applying a backward resolution-based step, which uses the rules in Σ as rewriting rules in the direction from right to left (*i.e.* from the rule head to the rule body). This rewriting step simulates an application of a TGD during the construction of the chase. In other words, after each rewriting step, we obtain a new query that is one level closer to the database-level, and when there is no applicable TGD, the obtained query reaches the database-level.

In the data exchange domain, there exists also a well-known algorithm called Chase and Back-chase (C&B) [33]. In a data exchange setting, we have a source schema S and a target schema T, where we assume that S and T are disjoint. Intuitively, the data transformation is done by using the relationship between the source and target schemas. Such relationship is called *source-to-target dependencies* that specify how and what source data should appear in the target. The C&B algorithm applies to the case when the source-to-target dependencies plays as the set of dependencies Σ in the query answering under TGDs problems. Indeed, C&B proceeds in two phases: *chase phase* and *backchase one*. In the chase phase, it uses Σ to chase the given conjunctive query until (and if) no more chase steps are possible. Rules in Σ are used as rewriting rules in the direction from left to right (*i.e.* from the rule body to the rule head). The result of this phase is a new conjunctive query, called universal plan. In the backchase phase, it searches in the finite space of the universal plan to eliminate its redundancies in all possible ways, thus obtaining minimal reformulations, i.e. reformulations containing no joins that are redundant under the constraints. The inspected subqueries are checked for equivalence (under the constraints) to the original query. This check is performed using the chase procedure, which in essence adds to a query redundant joins that are implied by the constraints.

It is worth noting that although our approach is inspired from the reformulating query step of PerfectRef and other approaches based on PerfectRef, we adopt a fundamentally different point of view. Instead of using rules from *right to left*, *i.e.* verifying the applicable condition between atoms in query body and the head of the rule (right hand side) then rewriting the query with the body of the rule (left hand side), our proposal uses rules in the opposite way, from *left to right*. More precisely, the body of rules is taken into account to verify the applicable condition with atoms in the body query, then the reformulation is done by using the head of rules. Indeed, from the point of view of "Knowledge Base", a rule $\phi \rightarrow \psi$ has a semantic that if ϕ holds in the KB then it can be inferred that ψ holds too. In this context, a query is evaluated against not only the asserted facts as ϕ in ABox but also the inferred facts as ψ. Thank to the chase procedure, ABox is supplemented with such inferred facts. Thus, the idea beyond the reformulation in PerfectRef is to add into the query the **knowledge** of rules (by which new facts are inferred). Thereby without saturating the ABox, the rewritten queries can obtain not only ground facts in Abox but also induced facts inferred by rules in TBox. Nevertheless,

from the Database point of view, a rule $\phi \rightarrow \psi$ states that if ϕ holds in the dataset then ψ *must hold* in the dataset. It means that, in our context, we must verify the presence of ψ in the dataset. If ψ does not hold, we must disregard ϕ in the answering query process. Thus, the idea beyond the chase in this context is to add into the query the **condition** to be verified stated by the rules. Thereby, the answers of rewritten queries satisfy the rules.

In other words, in our context where the constraint rules are used as database constraints on the query answers, query rewriting is used in order to restrict the results, generally resulting in fewer answers. Whereas the *PerfectRef* is used in order to expand the results, with the TBox rules used as inference rules, generally resulting in more answers.

2.2 Principles of Semantic Data Integration

2.2.1 Data integration principles

Data integration is the problem of combining the data residing in different sources, and providing the user with a unified view of this data. Through the provision of such a unified view so-called *global schema*, data integration may alleviate users from the knowledge on where data are, how data are structured at the sources, and how they can be accessed. Such global schema represents the intentional level of the integrated and reconciled data, and provides the elements for expressing the queries over the data integration system. In a nutshell, a data integration system is defined as follows [56]:
A data integration system $\mathcal{J}$ is a triple $\mathcal{J} = \langle \mathcal{G}, \mathcal{M}, \mathcal{S} \rangle$, where:

- $\mathcal{G}$ is the expected global schema, which provides both a conceptual representation of the application domain, and a reconciled, integrated, and virtual view of the underlying sources.

- $\mathcal{S}$ is the source schemas, i.e. schemas of the sources where data are stored.

- $\mathcal{M}$ is the mappings between $\mathcal{G}$ and $\mathcal{S}$, i.e. a set of assertions establishing the connection between the elements of the global schema and those of the source schema.

Two data integration architectures exist, namely the *data warehouse* and the *mediation* approaches. In the first one, the global schema is used for driving the (*i*) extraction of distributed data from their local sources, (*ii*) their transformation into a common structure and (*iii*) their loading in a centralized data warehouse. In the data warehouse approach, the data is extracted from the data sources, transformed, and loaded in the warehouse ahead of query time. Thus, the data is queried at the warehouse during the query time, but not at the data sources. When it can rely on an efficient database management system, the main advantage of this solution is the efficiency of query evaluation, as all sources data are physically stored in

a single repository. Therefore, it is typically preferred for very complex queries, e.g., for data mining. Its main drawbacks are its cost in terms of storage, and in terms of refreshment when updates performed on the original data sources must be propagated to the warehouse.

In the mediation approach, data are kept in sources and information is retrieved dynamically from original databases, at query time. The integration is virtual, in the sense that data stay in sources, but the user who interacts with the mediator, via the global schema, feels like interacting with a single database. The main drawbacks of this solution are related to its query-answering process, which is more complex compared to warehouse approach, where queries are posed to one single centralized repository. Clearly, in the mediation approach, when a query q_g is posed in terms of the global schema $\mathcal{G}$, the system must reformulate it in terms of a suitable set of queries q_s posed to the sources, send each computed sub-query q_{si} to the involved source $\mathcal{S}_i$, and compose the received results into a final global answer for the user. In this process, the *semantic mapping* that specifies the relationships between the schemas of the data sources and the global schema plays an essential role in the reformulation step. In the literature, there exist two well-known approaches for building the semantic mapping: Global-As-View (or simply GAV) and Local-As-View (or simply LAV). Mediation solutions naturally fit the open and distributed web context.

Both of the data warehouse and mediation approaches require the design of a shared global schema: semantic web ontologies can play this role. The notion of ontology used here, which represents the model and the structural framework for formalizing and organizing knowledge in the semantic web level, was defined by Thomas Gruber [44] as a "formal, explicit specification of a shared conceptualization". When semantic web and data integration are combined for overcoming semantic heterogeneity in order to share and efficiently reuse data among autonomous interconnected stakeholders, the integration paradigm is called *Ontology-Based Data Integration* [28].

2.2.2 Ontology-Based Data Integration

The architecture of an OBDI system consists of two levels: (i) the mediator receiving the user's queries, and (ii) the distributed sources that respond to these queries. The mediator, on the one hand, is composed of a global schema that provides an integrated view of all the sources and a vocabulary that can be used by users to create queries and, on the other hand, has information about the data on the sources by which the semantic mapping $\mathcal{M}$ is built. The sources contain the actual data. Each can be represented by an individual schema and is equipped with a wrapper that acts as an intermediary between the source and the mediator. The roles of these adapters are to adapt the sub-queries sent by the mediator, in terms of its query language, in the query language accepted by the source, and to translate the result returned by sources into the mediator's data model.

As mentioned above, in a OBDI system, the connection between the mediator

and the sources is established through the *semantic mapping* $\mathcal{M}$ defined between the global schema and the local schemas. This mapping is crucial to the operation of the system. Depending on how one wants to build the system, there are two well-known approaches that can be adopted to implement the *semantic mapping* $\mathcal{M}$: GAV and LAV. Thus, if the GAV approach is applied, the mediator has a set of correspondences established between the elements of the sources and those of the global scheme. On the other hand, if the LAV approach is implemented, the mediator in this case has an abstract description, in the form of views, of the content of each source. Besides GAV and LAV, some hybrid solutions, so-called GLAV (Global-Local-As-View), have also been defined and used [56, 17].

In LAV approach, content on local sources are described by a set of views on the global schema. Formally, if $\mathcal{J} = \langle \mathcal{G}, \mathcal{M}, \mathcal{S} \rangle$ is an ODBI system, the *semantic mapping* $\mathcal{M}$ established between $\mathcal{G}$ and $\mathcal{S}$, according to the LAV approach, are a set of assertions, each of which is of the form:

$$s \rightarrow q_{\mathcal{G}}$$

Where s is a source of $\mathcal{S}$ and $q_{\mathcal{G}}$ is a query on the global scheme $\mathcal{G}$. More concretely, in the LAV approach the content of each source $s \in \mathcal{S}$ must be characterized in terms of $q_{\mathcal{G}}$ views formulated on the global scheme [56]. The main advantage of this approach is that it favors the extensibility of the system, i.e., adding and removing sources imply neither any changes on the global schema nor any changes on mappings that do not concern the sources involved. In other words, when a new source is added, the mappings are enriched with new assertions, while in the case when a source is removed, only assertions involving that source are deleted from the mappings. However, the disadvantage of the LAV approach is that the processing of requests is complex to implement. Nevertheless, well-established algorithms exist to do it, in particular the famous MiniCon algorithm [75]. It is interesting to notice that in [67], the authors adapt traditional LAV query systems, such as the MiniCon, to fit the semantic web features. They develop a technique called SemLAV to process SPARQL queries over a LAV mediator, which execute the query on a partial instance of the global schema dynamically loaded from the relevant local views during the query evaluation. In this way they avoid the rewriting phase. Their experiments show that this technique outperforms traditional solutions.

Different from LAV, the GAV approach defines the global schema in terms of source schemas. Formally, if $\mathcal{J} = \langle \mathcal{G}, \mathcal{M}, \mathcal{S} \rangle$ is an ODBI system, the *semantic mapping* $\mathcal{M}$ established between $\mathcal{G}$ and $\mathcal{S}$, according to the GAV approach, are a set of assertions, each of which is of the form:

$$g \rightarrow q_{\mathcal{S}}$$

Where g is an entity of the global schema $\mathcal{G}$ and $q_{\mathcal{S}}$ is a query on the sources $\mathcal{S}$. More concretely, in the GAV approach, the content of each element g of the global schema must be characterized in terms of views $q_{\mathcal{S}}$ expressed on the sources [56]. In other words, GAV mappings are used to indicate the system how it can access the data when evaluating the different elements of the global schema. The main

advantage of this approach is that it facilitates the processing of requests. The mappings are indeed quite explicit to facilitate a translation of a user's queries. It is only necessary to replace the global predicates of the query with their local definitions. However, one of the limitations of the GAV approach is that the sources must be stable and known in advance. In addition, adding and removing sources are difficult to manage. Indeed, since the mapping definition of an entity in the global schema is composed of a combination of local entities from different sources, thus any change in the source schema requires a whole check these definitions to detect all necessary modifications.

In general, there exist three main ontology-based data integration architectures traditionally used [91, 28], namely:

- *The single-ontology approach*, where all the data sources are related to a global ontology, which defines the basic terms of the domain. This is the simplest approach, when all the sources have the same view of the domain. However, in the presence of sources with a different view of the domain, finding a consensus in a minimal ontology commitment may be a difficult task.

- *The multiple-ontologies approach*, where each data source is described with its own (local) ontology, as a peer-to-peer system. Though this approach is more flexible than the single ontology approach, it requires the construction of mappings between the local ontologies. The lack of a common vocabulary between sources can make this task difficult.

- *The hybrid approach*, which combines the two previous ones. This approach involves the use of local ontologies that subscribe to a common top-level vocabulary, alleviating thereby the definition of inter-ontology mappings. This approach can be costly, however, for the construction of a global shared vocabulary in addition to local ontologies [29].

2.3 Dealing with Inconsistent Databases

Constraints are semantic conditions that a database should satisfy. They capture an important part of the semantic of a given application. Typically, database management system checks the satisfaction of constraints to maintain the consistency of data and avoid the violation of constraints. When dealing with a huge amount of data, constraint satisfactory cannot always be guaranteed. In such a scenario data is usually spread in many different databases, organized by different management systems, each one having thus own characteristics. For instance, a system may not support thoroughly procedures of checking constraints (for example: a legacy system), or some constraints may be not supported for overpriced reason. In database systems having long running activities or workflows, the constraints may stop hold temporarily, to be restored in a further process. Inconsistency of databases often happens in integration data systems, where even if the data at each autonomous source satisfy local constraints, global constraint violation exists.

Query answering over inconsistent databases may lead to unexpected or meaningless answering. To overcome this obstacle, two different strategies have been proposed in the research community. The first one bases on the idea that only when the data no longer conflict with the constraints, queries can be answered. From this point of view, it is obvious that the data needs to be repaired, i.e. explicitly modifying the data in order to eliminate violation of constraints. This is the most direct strategy, but the explicit repair of data is not always convenient, or possible. Possible causes are connected to the policy of sources not allowing or restricting the direct modification on data; or simply because it is impossible to repair a huge amount of data, which is one of five V's challenges in Big Data. Another common application scenario that is difficult to apply this strategy is the data integration application, which provide a unified, virtual view of a set of autonomous information sources.

The second perspective bases on the idea that inconsistency is considered as a natural phenomenon in realistic settings, i.e. accepting the inconsistent data, leaving the data unchanged and trying to obtain only meaningful answers when evaluating queries. From this point of view, there are two common sub-strategies. The first one is to query the data regardless the constraints, then to apply the constraints on the answer set of the query to filter the answers satisfying the constraints. The contribution of the thesis follows this direction. An alternative strategy is the one called *consistent query answering*, which is based on the principle *schema is stronger than data* [42], i.e. the set of constraints (i.e. the schema) is considered as actually reliable information, while data are considered as information to be revised.

In consistent query answering approaches, in order to obtain the meaningful answers to a query, one try to determine the part of the database that is consistent with respect to all the constraints, called *repair* [55]. Indeed, a repair of a database contradicting a set of integrity constraints is a database obtained by applying a minimal set of changes to restore the consistency, i.e. it is consistent and minimally differs from the original database. There are many possible repairs for the same database, therefore computing the consistent answers to a query in an inconsistent database amounts to computing the tuples that are answers to the query in all possible repairs. Approaches in the literature on consistent query answering deal with different kinds of constraints in computing the consistent answer. Andrea Cali et al. in [22] propose a method for consistent query answering under key dependencies and exclusion dependencies based on the rewriting of the query in $Datalog^{Neg}$, a well-known extension of Datalog that allows for using negation in the body of program rules. Their idea is to convert key dependencies and exclusion dependencies into appropriate rules of a $Datalog^{Neg}$ program such that each stable model logic of the program is a maximal subset of tuples that are consistent with all the dependencies. This step bases on an algorithm, called ID-rewrite, which is based on the PrefectRef algorithm, to compute a perfect rewriting of a union of conjunctive queries Q. Rosati et al. in [55] extend the notion repair in Description Logic knowledge bases, called ABox Repair (AR). Then instead of computing all the possible repairs of the knowledge bases relevant for consistent query answering,

they propose the notion called Intersection ABox Repair (IAR) that calculates the intersection of such repairs as the ABox to use in query answering. The solution in [55] deals with $DL - Lite_{A,id,den}$ that corresponds to inclusion dependencies, key dependencies and denial dependencies. Proposal of Lukasiewicz et al. in [58, 57] is similar to [55]. It also proposes the notion like IAR, but applies to a fragment of Datalog+- that comprises linear tuple-generating dependencies, negative constraints and non-conflicting equality-generating dependencies. These dependencies are not exactly the same as the ones in [55]. For instance, the key dependencies (a.k.a. identification constraints) in [55] do not cover the restricted form of equality-generating dependencies considered in [58].

2.4 Conclusion

Although inconsistency of knowledge bases is also the kernel of works in [22, 42, 55, 58, 57, 81, 80], our proposal differs from them not only on the meaning given to constraints but also on the treatment imposed to detected inconsistencies.

In other words, the focus of these works contrasts with ours in the following way:

- Works such as [22, 42, 58, 57, 81, 80] focus on *consistency of the data stored in the source databases*. Indeed, in those works, given an inconsistent database, the general idea is to restore database consistency through the computation of a new consistent database or to refuse the database when consistency is not achievable without updates. In other words, without modifying the original base, their proposals try to use rules in the TBox to compute the new consistent database. This computation may imply materialization of facts which were originally implicitly defined or the consideration of different possible consistent states of the database.

- Our work focuses on the *consistency of the query answers returned to a user*. Instead of trying to establish the consistency of a database in order to query over it, we want to, firstly, just evaluate a query on a given (possibly) inconsistent database, and, then, use rules (which define a context) as filters to obtain the valid answers. In this way, we also obtain *consistent data*. Our goal is neither to modify the database nor to try to compute its different possible consistent states. We do not want to refuse an inconsistent database: since the meaning of consistency is in connexion to a user profile; one can admit things that are prohibited by others. Checking consistency of the answers is usually a smaller task than verifying the whole database validity.

Besides the focus, the meaning of the constraints is another aspect that distinguishes our approach from the above cited works.

- The cited works deal with ontological constraints, *i.e.*, inference rules capable of generating new facts from stored ones. Their role is therefore to define data, implicitly.

- Rules defining a context are, in our proposal, quality constraints. They impose a filter on retrieved data. Indeed, in our work constraints are seen in a database perspective (in contrast to the ontological perspective, where constraints are just inference rules). Notice, however, that our system can mix the above perspectives and deal not only with constraints (as explained above) but also with inference rules. This is possible when they are handled relatively independent and at different levels. More precisely, we can consider quality constraints at the top level of our system while ontological constraints can be used to infer knowledge on a lower level, from data sources.

From a technical point of view, as it will be clear in the next chapter, [33] presents a solution that is close to our work. However, it does not comprehensively consider the presence of constants in constraints and queries (note that constants are commonly used in real life scenarios). Therefore, the condition by which a constraint can be used as rewriting rules differs from ours.

On the other hand, [33] does not handle negative constraints, and the way it deals with equality-generating dependencies (EGD) differs from ours. Precisely, in [33] an equality is added to the rewritten query only when the whole body of an EGD matches a subset of atoms in the query's body. This is an EGD chase step [72]. Whereas, in our solution, an EGD has to be considered even if there is only one atom in EGD's body matching an atom in the query's body. For example, consider an EGD and a conjunctive query as follows:

$$\sigma : headOf(Xid, Xorg1), worksFor(Xid, Xorg2) \rightarrow Xorg1 = Xorg2$$

$$q(X,Y) \leftarrow Professor(X), worksFor(X,Y).$$

In this case, the EGD σ is taken into account in our solution, but it is ignored in the reformulation process proposed in [33]. Details of our verification of EGDs will be presented in the next chapter.

Part II

Contributions

Chapter 3

A Constraint-Based Query System

As described in the opening chapter, we want to develop a query system in which users can personalize the query context with different tools such as a set of constraints, a set of confidence degrees for data sources, etc. This context is then used as a filter to obtain context-driven answers from the answers of queries returned from semantic web data providers. In this chapter, we present in detail the system and how it works. In particular, we precise syntax and semantics of user quality constraints, the query answering over data sources with provenance, as well as the definitions of valid answers in this context. We then propose two methods of computing those context-valid answers: the naive approach based on validating auxiliary sub-queries and the query rewriting approach based on reformulating query by using constraints.

3.1 Querying environment

Our query processing system is depicted in Figure 3.1. It comprises two distinct parts which communicate: *Data validation*, responsible for checking constraints satisfaction, and *Data providers* for computing answers to the queries issued from the data validation part. The latter may actually integrate several end-data-providers, or it may connect only one provider. For ensuring that the final answers to the user's queries satisfy all user constraints, a dialogue between the two parts is established, for getting intermediate results and sending subsidiary queries.

The user defines his *query context* which is defined by a set of *global predicates* (given as the global schema over which queries can be built) and a set of *global constraints* on them (defining the quality requirement imposed by the user). These "global" constraints impose restrictions on answers coming from data providers. They filter information not respecting a constraint. The user's *query* involves global predicates, so quality constraints can be used to reformulate each query q, resulting in a set of conjunctive queries. Their answers not only are contained in q's answers but also are valid with respect to the user quality constraints.

Afterwards, these rewritten queries are sent to the *Data providers* part, which evaluates them against data stored on sources. The query evaluation process is transparent to the validation step. A query, resulting from the validation step, should be evaluated on the basis of providers' constraints and data. Indeed, each

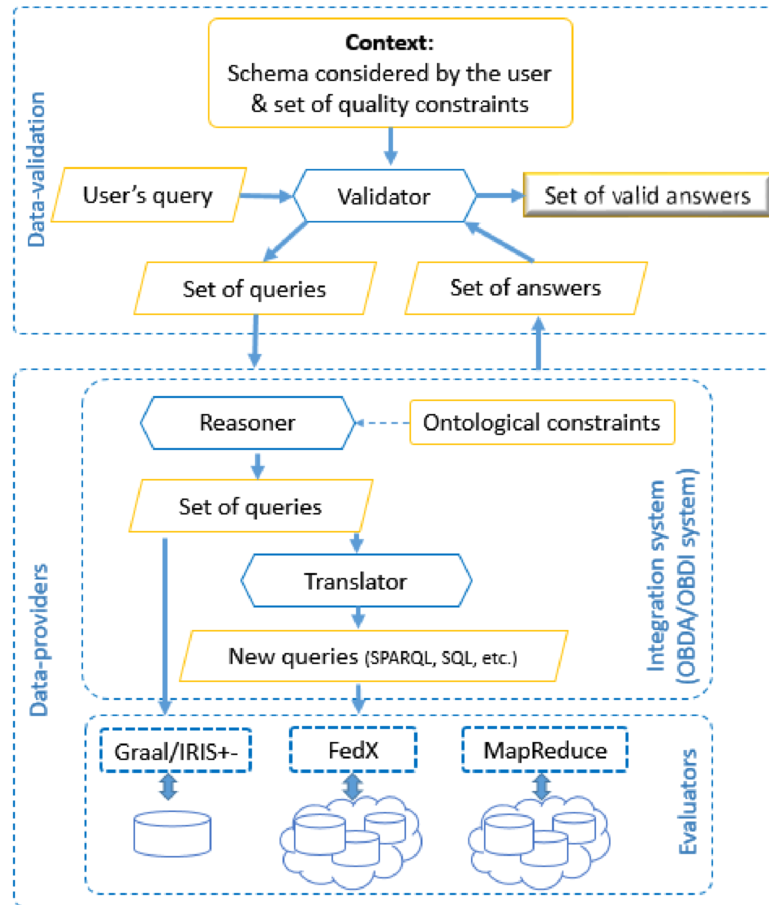

Figure 3.1 – Query system overview

provider may not only offer a set of data but also may impose its own ontological constraints allowing data inference (which relies on Open World Assumption). Ontological constraints are used as rewriting-rules to reformulate a query into a set of new conjunctive queries, for taking into account integration information (OBDA/OBDI Systems [74, 5]), or for dealing with incomplete information issues [5, 39, 55, 40]. But such rewritings are performed by the *Data providers* part, independently from the *Data validation* part.

The rewritten queries are evaluated actually at end-data-providers, called *local data sources*. The system can integrate a set of distributed local sources, or it may connect only one source. Note that these local sources can be equipped with various capabilities as well as support different query languages. Thus, the translator module is responsible to translate rewriting queries in the form of datalog into other query languages in case of necessity. For instance, FedX [87] is a federated system that allows evaluating queries of our system against data stored on a set of local sources. FedX only supports SPARQL. As a result the translator must convert queries from datalog into SPARQL. In case of dealing with huge data in parallel manner, our system can interact with MapReduce system where some

query languages such as HIVE-SQL or Pig Latin are supported.

In reality, information retrieved from data sources has different accurate levels that depend on many factors such as the origin of information, policies for updating and maintenance , and so on. For example, the information about the career or address of a person can be very different on data sources such as LinkedIn, Facebook, databases of his companies, home page, etc. Thus, user may determine the accuracy of information on each data source, and then associate them with corresponding confidence degrees. The set of confidence degrees of local sources together with the set of global constraints constitute the user's context in our system.

Motivating example In order to illustrate our system, we introduce a running example.
We consider a university database that contains a set of the following global predicates: $professor(X_{id})$ and $employeeGov(X_{id})$ indicates whether a person is a professor or only an employee working for the government; $teacherOf(X_{id}, X_{course})$ associates a teacher and the course he teaches, while $takesCourse(X_{id}, X_{course})$ refers to a student enrolling in a course; $offeredCourseAt(X_{course}, X_{dep})$ indicates in which department a course is taught; $researchesIn(X_{id}, X_{domain})$ and $worksFor(X_{id}, X_{dep}, X_{org})$ refer to the research domain and the work place (a department of an organization) of a person; $headOf(X_{id}, Y_{dep})$ indicates the head of a department.

Basing on the set of global predicates, called global schema, user defines global constraints. In this example, suppose we have a set of constraints of the Table 3.1

$\mathcal{C}_P$	POSITIVE CONSTRAINTS
c_{P_1} :	$professor(X_{id}) \rightarrow teacherOf(X_{id}, X_{course})$.
c_{P_2} :	$teacherOf(X_{id}, X_{course}) \rightarrow offeredCourseAt(X_{course}, X_{dep})$.
c_{P_3} :	$professor(X_{id}) \rightarrow employeeGov(X_{id})$.
c_{P_4} :	$teacherOf(X_{id}, DB) \rightarrow researchesIn(X_{id}, DB)$.
$\mathcal{C}_N$	NEGATIVE CONSTRAINTS
c_{N_1} :	$teacherOf(X_{id}, X_{course}), takesCourse(X_{id}, X_{course}) \rightarrow \bot$
$\mathcal{C}_K$	KEY CONSTRAINTS
c_{K_1} :	$worksFor(X_{id}, X_{dep}, X_{org}), headOf(X_{id}, Y_{dep}) \rightarrow X_{dep} = Y_{dep}$

Table 3.1 – Set of constraints on $\mathbb{G}$

whose meaning is indicated below:

- c_{P_1} : every professor has to offer a course.
- c_{P_2} : every course given by a teacher must be associated with a department.
- c_{P_3} : professors are government employees.
- c_{P_4} : if a teacher offers a database course, then he must be a researcher in the database domain.

- c_{N_1} : nobody can teach and register in a same course.
- c_{K_1} : the head of a department in an organization cannot be a person working in a different department.

The global schema is also used to build user's queries, and their results are computed from data coming from a distributed database, composed by our so-called local data sets. Table 3.2 illustrates local sources which are not trusted equally by the global system. A confidence degree (τ) indicates the accuracy associated with each one. With these source confidence degrees the entire querying context is settled. Source 1 is considered to be accurate (95% reliable) while the reliance on Source 3 is smaller (accuracy: 70%).

Source 1, $\tau_{S1} = 0.95$	**Source 2**, $\tau_{S2} = 0.80$	**Source 3**, $\tau_{S3} = 0.70$
professor(Bob)	offeredCourseAt(DB,LIFO)	professor(Peter)
professor(Tom)	offeredCourseAt(Java,LIFAT)	professor(Ann)
professor(Alice)	worksFor(Bob,LIFO,UO)	headOf(Bob,LIFO)
bornIn(Bob,USA)	worksFor(Ann,LIFAT,UT)	headOf(Ann,CNRS)
bornIn(Tom,UK)	takesCourse(Tom, Java)	bornIn(Peter,UK)
bornIn(Alice,Denmark)	takesCourse(Bob, Net)	bornIn(Ann,USA)
foreignCountry(USA)	teacherOf(Bob, DB)	teacherOf(Peter,Java)
foreignCountry(UK)	teacherOf(Bob, Java)	teacherOf(Ann,DB)
foreignCountry(Denmark)	teacherOf(Tom, Java)	researchesIn(Bob,DB)
	teacherOf(Alice, Net)	employeeGov(Bob)
		employeeGov(Tom)
		employeeGov(Alice)
		employeeGov(Peter)

Table 3.2 – Example of local sources

Suppose one user wants to work in a context that contains only constraints c_{P1} and c_{P2}. In this scenario let us consider query

$$q(X) \leftarrow professor(X), bornIn(X,Y), foreignCountry(Y)$$

to find the foreign professors. The required confidence degree is $\tau_{in} = 0.75$, indicating that sources having a smaller confidence degree should not be taken into account. More precisely, in this case only data from the source 1 and source 2 are considered. The source 3 is ignored as $\tau_{S3} < \tau_{in}$. The answer is the set $\{(Bob) : 0.8, (Tom) : 0.8\}$ as both Bob and Tom are professor ($professor(Bob), professor(Tom)$); they were born in foreign countries ($bornIn(Bob, USA)$, $bornIn(Tom, UK)$, $foreignCountry(USA)$, $foreignCountry(UK)$), and they teach at least one course ($teacherOf(Bob, DB)$, $teacherOf(Tom, Java)$) at a department ($offeredCourseAt(DB, LIFO)$, $offeredCourseAt(Java, LIFAT)$). Tuple ($Alice$) is not an answer because although she offers a course "Net" ($teacherOf(Alice, Net)$) (i.e. c_{P1} is satisfied), her course is not associated with any department, thus it violates constraint c_{P2}.

Another user is stricter: he wants a context where all the constraints in Table 3.1 are used. He uses the same query of the first scenario but with required confidence degree $\tau_{in} = 0.7$, i.e. data from all three sources are considered to answer the query.

It is easy to see that the course *Net* of *Alice* is not associated with any department (violating c_{P2}), *Tom* teaches *Java* but he also registers in a course *Java* (violates c_{N1}), while *Ann* is not a government employee (violating c_{P3}). In fact, c_{P4} and c_{K1} are also violated as *Ann* teaches database but she does not do research in database domain. Moreover, she is the head of *CNRS* while not working there. Only *Bob* and *Peter* satisfy all constraints. Thus, the answer is the set $\{(Bob) : 0.7, (Peter) : 0.7\}$. □

3.2 Background

This section presents definitions and notations used in the rest of the chapter.

Alphabet and notations. Let $\mathbf{A}$ be an alphabet consisting of constants, variables, predicates, the equality symbol (=), quantifiers ($\forall$ and $\exists$) and the symbols $\top$ (true) and $\bot$ (false). We consider four mutually disjoint sets, namely:

- $\mathbf{A}_C$, a countably infinite universe set of *data constants*, called the underlying database domain;
- $\mathbf{A}_N$, a countably infinite set of fresh labeled *nulls* which are placeholders for unknown values, and can be seen as variables;
- VAR, an infinite set of *variables* that will be used to range over elements of $\mathbf{A}_C \cup \mathbf{A}_N$
- PRED, a *finite* set of *predicates* or relation names; each predicate is associated with a positive integer called its arity.

Since we consider a function-free language, the only possible *terms* are constants, nulls or variables.

Free Tuple. A *free tuple* u is a sequence of either variables or constants, or both. We denote by $var(u)$ the set of variables in the free tuple u.

Atomic formulas. An atomic formula (or *atom*) is constructed from predicate and terms, which has one of the forms:

- $P(t_1, ..., t_n)$, where P is an n-ary predicate, and $t_1, ..., t_n$ are terms;
- expressions $\top$ (true) and $\bot$ (false)
- $t_1 = t_2$, where t_1 and t_2 are terms.

A conjunction of atoms is often identified with the set of all its atoms.

Given a predicate $P \in$PRED of arity n, a fact over P is an atom $P(u)$ where $u \in (\mathbf{A}_C)^n$.

Substitution and homomorphism.
A *substitution* from the set of symbols E_1 to the set of symbols E_2 is a function $h : E_1 \rightarrow E_2$.
A *homomorphism* from a set of atoms A_1 to a set of atoms A_2 is a mapping h from the terms of A_1 to the terms of A_2 such that:

- if $t \in A_C$, then $h(t) = t$;
- if $t \in A_N$, then $h(t) \in (A_C \cup A_N)$;
- if t is $\perp$, then $h(\perp) = \perp$;
- if t is $\top$, then $h(\top) = \top$;
- if $r(t_1, ..., t_n)$ is in A_1, then $h(r(t_1, ..., t_n)) = r(h(t_1), ..., h(t_n))$ is in A_2;
- and h naturally extends to sets of atoms and conjunctions of atoms.

Isomorphism and Unification.
An *isomorphism* is a bijective homomorphism, *i.e.*, the set of atoms A_1 is isomorphic to the set of atoms A_2 iff there exists a homomorphism h_1 from A_1 to A_2 which admits a homomorphism from A_2 to A_1.
An *endomorphism* h on A_1 is a homomorphism such that $h(A_1) \subseteq A_1$.
Two atoms A and A' are *unifiable* iff there exists an endomorphism σ (denoted by *unifier*) on $\{A, A'\}$, such that $\sigma(A) = \sigma(A')$. If A and A' are unifiable, then they have a unifier θ such that every unifier σ of A and A' can be written as $\sigma = h\theta$ for some endomorphism h on $\{A, A'\}$. Such a unifier is called the *most general unifier* of A and A' (denote by *mgu*).

In this thesis, we consider a global database schema $\mathbb{G}$ *i.e.* a set of predicate symbols, over which user's context and query are built.

Definition 3.1 (Rule) A *(datalog) rule* r has the form

$$R_0(u_0) \leftarrow R_1(u_1) \ldots R_n(u_n), comp_1(v_1), \ldots, comp_m(v_m)$$

where $n, m \geq 0$, R_i $(0 \leq i \leq n)$ are predicate names, u_i are free tuple of appropriate arity and $comp_j(v_j)$ $(0 \leq j \leq m)$ are comparison formulas having the form $(X = a)$ or $(X = Y)$ where X and Y are variables appearing in u_i $(0 \leq i \leq n)$ and a is a constant.

The head of r (denoted by $head(r)$) is the expression on the left hand-side of the rule while the right hand-side is denoted by $body(r)$. The *arity* of the rule is the arity of $head(r)$, *i.e.* the arity of R_0 . Rules are: (a) *range restricted*, *i.e.* only variables appearing in the rule body can appear in the head and variables of comparison formulas are variables in atoms $R_i(u_i)$ and (b) *satisfiable*, *i.e.* the transitive closure of the comparison formulas in the query does not lead to contradictions such as requiring two different constants to be equal. We recall from [4] that each satisfiable rule with equality is equivalent to a rule without equality.

Definition 3.2 (Query) A *conjunctive query* (CQ) q of arity n over a given schema is a rule of arity n. A *boolean conjunctive query* (BCQ) is a conjunctive query of arity zero, denoted as *q()*.

Answers to queries Let I be an instance (*i.e.* a set of facts) for a given schema. The *answer* to a conjunctive query q of arity n over I, denoted as $q(I)$, is the set of all n-tuples $t \in \mathbf{A}_C^n$ for which there exists a homomorphism h_t such that $h_t(body(q)) \subseteq I$ and $h_t(u_0) = t$. We denote by h_t *a homomorphism used to obtain an answer tuple* t. Technically, the answer *false* (*i.e.*, a negative answer) for a BCQ corresponds to the empty result set and the answer *true* (*i.e.*, a positive answer) corresponds to the result set containing the empty tuple. A positive answer over I is denoted by $I \models q$.

A union of conjunctive query (UCQ) Q of arity n is a set of conjunctive queries, where each $q \in Q$ has the same arity n and uses the same predicate symbol in the head. The answer to Q over an instance I, denoted as $Q(I)$, is defined as the set of tuples $\{t \mid \text{ there exists } q \in Q \text{ such that } t \in q(I)\}$.

Example 3.1 Considering the query in the above running example:

$$q(X) \leftarrow professor(X), placeOfBirth(X,Y), foreignCountry(Y)$$

It is a conjunctive query with $arity = 1$. If we consider only data from the Source 1, then the corresponding homomorphisms used to obtain answer tuples are

$$(h_1(X) = Bob,\ h_1(Y) = USA)$$
$$(h_2(X) = Tom,\ h_2(Y) = UK)$$
$$(h_3(X) = Alice,\ h_3(Y) = Denmark)$$

3.3 Graph database, constraints and provenance

As mentioned in the Section 3.1, our query environment is composed of two independent parts that are the Data validation part and the Data providers part. In this context, the following aspects of our approach are worth underlings:

1. Let q be a query over the global schema $\mathbb{G}$. Answers for q are filtered according to quality restrictions settled for an application. A user may establish the context in which a certain number of queries is evaluated and choose another context for other queries. The customization of this quality context is provided by a set of constraints $\mathcal{C}$ on the global schema $\mathbb{G}$. Only data respecting them is allowed as query answers. The query q is rewritten to take into account constraints $\mathcal{C}$ imposed by the settled context. The result is a set of queries $\mathcal{Q}$ that is sent to the *query evaluation*. Inconsistencies on sources are allowed and our approach does not aim at correcting them, but at discarding them during query evaluation.

2. We may see data providers as local data sources $(\mathcal{S}_1, \tau_1), \ldots, (\mathcal{S}_n, \tau_n)$ storing a distributed graph instance. For $1 \leq i \leq n$, each $\mathcal{S}_i$ is a data source instance respecting a local schema $\mathbb{S}_i$ and each τ_i is the source confidence degree, represented by a number in the interval $[0, 1]$.

3. We assume the existence of a mapping between global and local systems (*e.g.* via a LAV approach). In general, when a global query q has a non-empty set of answers then there exists at least one re-writing of q in terms of sub-queries $q'_1 \ldots q'_m$ where each q'_j $(1 \leq j \leq m)$ is a sub-query expressed over local relations.

4. Our proposal can be implemented over different evaluation mechanisms. When confidence degrees are disregarded we write $\overline{\mathcal{S}} \models q$, a shorthand of $\mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_n \models q$, to denote that the answer of a BCQ q is positive with respect to local databases. In this way, we see local databases as a whole, *i.e.*, a system (or distributed database) capable of answering our global query. We denote by $ans(q, \overline{\mathcal{S}})$ the set of tuples obtained as answers for a conjunctive global query q over the distributed database $\overline{\mathcal{S}}$.

5. When data provenance is considered, we associate a confidence degree to each source and a minimal confidence degree (τ_{in}) to the query. These measures are settled by the user, according to his knowledge of source accuracy or proposed by a recommendation system. Our system offers some flexibility in how to use veracity information.

   - A query is associated with a minimal confidence degree (τ_{in}).
   - The system offers the possibility to access only sources respecting a given condition with respect to τ_{in} (*e.g.* , sources having a confidence degree superior to τ_{in}) or to take all sources into account.
   - The confidence degree (τ_{S_i}) of each source S_i accessed in order to compute q's answer is used in the computation of an output confidence degree (τ_{out}) according to a calculation f which can be proposed by the user (*e.g.* , the average of the concerned confidence degrees).

   As a natural extension of item 4 above, we write $\overline{(\mathcal{S}, \tau)} \models q : \tau_{in}$ and we denote by $ans(q : \tau_{in}, \overline{(\mathcal{S}, \tau)})$ the set of tuples obtained as answers for a conjunctive global query $q : \tau_{in}$ over a distributed database $\overline{(\mathcal{S}, \tau)}$.

6. Once answers for $\mathcal{Q}$ are computed on the basis of data providers, they are sent to the validator that still verifies validation with respect to constraints not used during the rewriting-step in 1. Subsidiary queries may be generated in this second validation phase establishing a validation-evaluation dialogue. Once this dialogue terminates, all constraint verification is done. Valid answers are sent to the user.

The following definition introduces the constraints used in our environment.

Definition 3.3 (Constraints) We define a user *context* as a set $\mathcal{C}$ of constraints, composed of three subsets, as follows:

• ***Positive constraints*** ($\mathcal{C}_P$): Each positive constraint has the form
$$c : L_1(u_1) \rightarrow L_2(u_2)$$
where c is the name of constraint, L_1 and L_2 are predicate (or relation) names in $\mathbb{G}$; u_1 and u_2 are free tuples (*i.e.*, may use either variables or constants) having appropriate arities.
Assuming that $var(u_1) \cap var(u_2) = \{x_1, \ldots, x_k\}$ and that $var(u_1) = \{x_1, \ldots, x_k, x_{k+1}, \ldots, x_n\}$, $var(u_2) = \{x_1, \ldots, x_k, y_1, \ldots, y_m\}$ where ($n \geq k \geq 0$, $m \geq 0$) then a positive constraint can be represented in the following form:
$$c : \forall x_1, \ldots, x_n \quad L_1(u_1) \rightarrow \exists y_1, \ldots, y_m \quad L_2(u_2)$$

• ***Negative constraints*** ($\mathcal{C}_N$): Each negative constraint is a rule where all variables $u^1, \ldots, u^k$ in the free tuple u are universally quantified and which has the form:
$$c : \phi(u) \rightarrow \bot$$
Formula $\phi(u)$ is an atom $L_1(u)$ or a conjunction of two atoms $L_1(u_1), L_2(u_2)$, for which if $var(u_1)$ and $var(u_2)$ are both non-empty sets, then $var(u_1) \cap var(u_2) \neq \emptyset$, *i.e.* if there exist variables in u_1 or u_2, then L_1 and L_2 must have at least one common variable. We also use the negative constraint counterpart with equalities in *comp*:
$$c : \phi'(u), comp \rightarrow \bot$$
equivalent to the previous one, but explicitly expressing *equalities* between variables and/or constants. We refer to $\mathcal{C}_{N1}$ and $\mathcal{C}_{N2}$ as sets of negative constraints having only one atom and two atoms, respectively.

• ***Equality-generating dependency constraints*** without nulls ($\mathcal{C}_K$) (also called key constraints): Each EGD is a rule having the general form
$$c : \quad L_1(u_1), L_2(u_2) \rightarrow u'_1 = u'_2$$
where all variables in the free tuple u_1 and u_2 are universally quantified; L_1, L_2 are predicate names in $\mathbb{G}$; u_1, u_2 are free tuples such that $var(u_1) \cap var(u_2) \neq \emptyset$; u'_1 and u'_2 are sub-tuples (*i.e.* an ordered-subset of variables) of var(u_1) and var(u_2) respectively.
Notice that EGD include functional dependency (and thus, key constraints) having the form: $L_1(u_1), L_1(u_2) \rightarrow u'_1 = u'_2$ □

For simplicity, we usually omit quantifiers, using the shorthand format of constraints.
In this paper, we consider that the set of positive constraints is weakly acyclic [35], which ensures that positive constraints "do not have a cyclic condition such that another null value forces the adding of a new null value" [72].

Example 3.2 The positive constraint *"If a teacher gives a course then there exists a department responsible for this course."* can be represented by the following formula:

$$c_1 : teacherOf(X_{id}, X_{course}) \rightarrow offeredCourseAt(X_{course}, X_{dep})$$

The constraint *"The head of a department must not teach "* is a negative constraint and its formula is:

$$c_2 : teacherOf(X_{id}, X_{course}), headOf(X_{id}, X_{dep}) \rightarrow \perp$$

The constraint *"Nobody can teach and register in Database course in same time"* is a negative constraint and its formula is:

$$c_3 : teacherOf(X_{id}, DB), takesCourse(X_{id}, DB) \rightarrow \perp$$

it is equivalent to:

$$c'_3 : teacherOf(X_{id}, X_{course}), takesCourse(X_{id}, X_{course}), (X_{course} = DB) \rightarrow \perp$$

The constraint *"A person who is the head of a department cannot work for a different department"* has form:

$$c_4 : worksFor(X_{id}, X_{dep}, X_{org}), headOf(X_{id}, Y_{dep}) \rightarrow X_{dep} = Y_{dep}$$

□

Given an atom A, in general a positive constraint c is triggered by an atom A when there is a homomorphism h from $body(c)$ to A. Positive constraints are a special case of linear tuple generating dependency (TGD [4]) which contain only one atom in the head. When the set of existential variables of c is not empty, the homomorphism h is extended to h' such that, for each existential variable z, $h'(z)$ is a new fresh variable. Clearly, the result of a positive constraint can trigger another positive constraint and so on.

It is worth noting that to compute the set of atoms generated by positive constraints we use the *chase procedure* (Section 2.1). In data exchange domain, the chase modifies an instance by a sequence of chase steps until all dependencies are satisfied - recalling that *a chase step corresponds to the result of triggering one constraint.*

In our approach, constraints are triggered on the basis of atoms in the query's body. Indeed the result returned by our chase procedure can be computed in two different ways which define two different methods of computing *context-valid* answers.

- In the naive approach, atoms in the query's body, instantiated by a homomorphism h_t used to produce an answer t to query q, are responsible for triggering constraints. In this case, the approach consists in obtaining an answer t and then checking by successive sub-queries whether t respects constraints in $\mathcal{C}_P$. Details of this approach are explained in Section 3.5.
- In the query rewriting approach, a chase procedure produces new atoms from those in the query's body to obtain a new query (or a set of queries having the same head) which incorporates the restrictions imposed in $\mathcal{C}_P$. Our method is similar to the universal plan proposed in [33] but introduces specificities concerning constraints involving constants. Section 3.6 presents this approach.

Now, before detailing the above methods, in the next section, we formalize the semantics of valid answers in our approach.

3.4 Valid query answer: formal definitions

As stated before, our querying system takes into account a personalized context together with the confidence degrees of data sources. When data sources are associated with confidence degrees, a query q can have a required confidence degree τ_{in} and one can expect that only data coming from sources whose confidence degrees respect a given condition with respect to τ_{in} are taken into account to build answers for q.

Definition 3.4 (Local querying with confidence) Let $(\mathcal{S}, \tau)$ be a local source database where $\mathcal{S}$ is a database instance and τ is the truth or confidence degree of the database. Let q be a query over $(\mathcal{S}, \tau)$ with the minimum required truth degree τ_{in}. The answer of $q : \tau_{in}$ over $(\mathcal{S}, \tau)$ is the set $ans(q : \tau_{in}, (\mathcal{S}, \tau)) = \{(t : \tau_{out}) \mid \tau_{out} = \tau$ and $t \in q(\mathcal{S})$ and $cond(\tau_{in}, \tau)\}$, where $cond(\tau_{in}, \tau)$ is a condition we may establish to avoid considering some sources. □

The system can be parametrized with other conditions, and even no condition can be settled at this step (*i.e.* all sources are considered in the computation of τ_{out}).

To compose the response to a query $q : \tau_{in}$, we put together answers produced by differently trusted databases. Firstly, we need a set of possible candidate answers: tuples t that are trustable with respect to τ_{in}.

Definition 3.5 (Candidate answer over $\overline{(\mathcal{S}, \tau)}$) Let $\overline{(\mathcal{S}, \tau)}$ be a graph database instance composed of n local databases having different truth degrees. A couple $(t : \tau_{out})$ is a candidate answer for a global query $(q : \tau_{in}, \overline{(\mathcal{S}, \tau)})$ if the following conditions hold: (1) tuple t is an answer obtained from local sources, *i.e.* $t \in ans(q, \overline{\mathcal{S}})$; (2) $\tau_{in} \leq \tau_{out}$ and the computation of τ_{out} is defined by $\tau_{out} = f(\tau_{in}, \{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$ where:
(i) each $\tau^k_{out_{S_j}}$ denotes the degree of the tuples in $ans(q_k : \tau^k_{in}, (S_j, \tau_j))$ for the sub-query q_k $(1 \leq k \leq m)$ generated to be evaluated on the local source (S_j, τ_j) during the evaluation process of q (where $i, j, l \in [1, n]$) and
(ii) function f computes a confidence degree taking as input the query confidence degree and the confidence degrees of data sources concerned by q. □

A user can parametrize the use of confidence degrees by choosing different functions f and by combining this choice with *cond* in Definition 3.4. For example, consider that in Definition 3.4 $cond(\tau_{in}, \tau) = true$. In this case, the selection of t is based *only* on the confidence degree computed by f. If f is the average, the resulting τ_{out} computes the average of *all* data sources involved in the query answering. If, however, a condition such as $\tau_{out} \geq 0.5$ is used in Definition 3.4, only sources respecting it are used in the average computation. Our examples consider that $f(\tau_{in}, \{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$ corresponds to $min(\{\tau^1_{out_{S_i}}, \ldots, \tau^m_{out_{S_l}}\})$, and as stated above, we disregard sources whose confidence is inferior to τ_{in}.

Global query answers are restrained by constraints in $\mathcal{C}$. To find an answer t to a query q means to find an instantiation h_t for the body of q that generates t. Verifying whether $h_t(body(q))$ is valid with respect to $\mathcal{C}$ ensures the validity of our answer. In the following definition we put together constraint and confidence degree verification to answer global queries.

Let $q{:}\tau_{in}$ be a conjunctive global query and $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_N \cup \mathcal{C}_K$ be a set of constraints over $\mathbb{G}$. Valid candidate answers are those that respect constraints and are obtained by trusted databases.

Notice that, in the following, given a set $\mathcal{C}_P$ of positive constraints, we denote by $chase(\mathcal{C}_P, I)$ the procedure capable of computing all consequences of $\mathcal{C}_P$ in an instance I. We only allow positive constraints respecting the weak acyclicity condition [35] - the first sufficient polynomial-time condition for checking if $\mathcal{C}$ has a terminating chase. We refer to [72, 12, 31] for details on the different chase algorithms.

Definition 3.6 (Valid candidate answers) The set of valid candidate answers of a query $q : \tau_{in}$, restrained by $\mathcal{C}$, over a database $\overline{(\mathcal{S}, \tau)}$, denoted by $valCandAns$ $(q{:}\tau_{in}, \mathcal{C}, \overline{(\mathcal{S}, \tau)})$ is defined by the set $\{(t : \tau_{out})\}$ respecting the following conditions:

- t is a candidate answer as in Definition 3.5 and h_t is a corresponding homomorphism
- there exists h_t such that for all $L \in h_t(chase(\mathcal{C}_P, J))$, where $J = \{l | l \in h_t(body(q))\}$, the following conditions hold:
  - there is a positive answer for $q() \leftarrow L : \tau_{in}$ on $\overline{(\mathcal{S}, \tau)}$;
  - for each $c \in \mathcal{C}_N$ of the form $L_1, L_2 \rightarrow \bot$, if there is a homomorphism ν such that $\nu(L_i) = L$, then there is no homomorphism ν' that extends ν and for which there is a positive answer for $q'() \leftarrow \nu'(L_{\bar{i}}) : \tau_{in}$ (In our notation, if $i = 1$ then $\bar{i} = 2$ and vice-versa);
  - for each $c \in \mathcal{C}_N$ of the form $L_1 \rightarrow \bot$, there is no homomorphism ν such that $\nu(L_1) = L$;
  - for each $c \in \mathcal{C}_K$ of the form $L_1(u_1), L_2(u_2) \rightarrow X_1 = X_2$ where X_1 and X_2 are variable in u_1 and u_2 respectively, if there is a homomorphism ν such that $\nu(L_i(u_i)) = L$, then the answer of $q(X_{\bar{i}}) \leftarrow \nu(L_{\bar{i}}(u_{\bar{i}})) : \tau_{in}$ is a singleton containing the tuple value $\nu(X_i)$.

As already mentioned, valid answers can be computed in different ways. The naive approach is a direct implementation of Definition 3.6 while the rewriting approach focuses on building new queries that incorporate the constraints. The following sections present each of these two methods.

3.5 How to obtain valid answers: the naive approach

The main idea of the naive approach is to validate each answer of q by generating sub-queries corresponding to each constraint $c \in \mathcal{C}$. Then based on the evaluation of all those sub-queries, we decide to accept or to reject the answer.

The process consists of three steps: (i) evaluate the initial query q to obtain a set of candidate answers $\mathcal{R}'$, (ii) generate suitable sub-queries from each answer in $\mathcal{R}'$ for each constraint $c \in \mathcal{C}$, and (iii) evaluate those sub-queries and decide the validity of the answer in the final result $\mathcal{R}$.

Given a query q and t is an answer to it, let h_t be the homomorphism used to produce tuple t. We want to check whether t is valid with respect to constraints. Tuple t is considered valid only when *all* constraints triggered during the validation process are satisfied. The naive validating method of a query q with respect to a set of user's quality constraints $\mathcal{C}$ is performed by Algorithm 4.

Let $L(\mathbf{X})$ be an atom of $body(q)$. The instantiated atom $h_t(L(\mathbf{X}))$ may trigger a constraint c. According to the type of c, a suitable sub-query q' is created:

- For $c \in \mathcal{C}_P$, q' is a boolean query: $q'() \leftarrow h_t(L_0(\mathbf{X}_0))$ where $L_0(\mathbf{X}_0) = head(c)$. The resulting tuple t is valid with respect to c if the answer of q' is positive. Notice, however, that each fact f resulting from the instantiation of $h_t(L_0(\mathbf{X}_0))$ on the database may trigger another constraint. The validation process continues until no constraint is triggered as stated by the use of h_1 in Definition 3.6, which corresponds to a chase procedure. This process establishes a dialogue between the validator and the providers. This is done in Algorithm 4 by the recursive call in line 10.

- For $c \in \mathcal{C}_N$ and assuming that c has the form $L(\mathbf{X}), L_0(\mathbf{X}_0) \rightarrow \perp$ the boolean sub-query is $q'() \leftarrow h_t(L_0(\mathbf{X}_0))$. Tuple t is valid with respect to c if the answer of q' is negative (line 16). Clearly, if c has the form $L(\mathbf{X}) \rightarrow \perp$, the verification is straightforward (line 12).

- For $c \in \mathcal{C}_K$, assuming that c has form $L(\mathbf{Y}, X_1, \mathbf{Z}_1), L_0(\mathbf{Y}, X_2, \mathbf{Z}_2) \rightarrow X_1 = X_2$ and $\mathbf{X} = \mathbf{Y} \cup X_1 \cup \mathbf{Z}_1$, the sub-query is $q'(X_2) \leftarrow h_t(L_0(\mathbf{Y}, X_2, \mathbf{Z}_2))$. Tuple t is valid with respect to c if the answer set is a singleton containing the tuple value $h_t(X_1)$ (line 20).

3.6 How to obtain valid answers: the query rewriting approach

In this section, we describe the query rewriting approach to tackle the problem of validating query answers with respect to constraints, whereby rewritten queries em-

Algorithm 4: Naive validation

Input : • A conjunctive query $q : \tau_{in}$ and a set of constraints $\mathcal{C} = C_P \cup C_N \cup C_K$.
• An access to the database instance $\overline{(\mathcal{S}, \tau)}$

Output: Answers of $q : \tau_{in}$ respecting $\mathcal{C}$.

1 **Function** `NaiveValidation(`$q : \tau_{in}, \mathcal{C}$`)`**:**
2 $\quad \mathcal{R} = \emptyset$;
3 $\quad$ **foreach** $(t, h_t) \in ans(q : \tau_{in}, \overline{(\mathcal{S}, \tau)})$ **do**
4 $\quad\quad valid = True$;
5 $\quad\quad J = h_t(body(q : \tau_{in}))$;
6 $\quad\quad$ **foreach** $L \in J$ **do**
7 $\quad\quad\quad$ **foreach** $c \in \mathcal{C}$ **do**
8 $\quad\quad\quad\quad$ **if** $c \in \mathcal{C}_P$ *and* $\exists h_1$ *such that* $h_1(body(c)) = L$ **then**
9 $\quad\quad\quad\quad\quad$ Let $q' : \tau_{in}$ be $q'() \leftarrow h_1(head(c))$;
10 $\quad\quad\quad\quad\quad$ **if** $NaiveValidation(q' : \tau_{in}, \mathcal{C})$ *is empty* **then**
11 $\quad\quad\quad\quad\quad\quad valid = False$;
12 $\quad\quad\quad\quad$ **if** $c \in \mathcal{C}_{N1}$ *and* $\exists h_1$ *such that* $h_1(body(c)) = L$ **then**
13 $\quad\quad\quad\quad\quad valid = False$;
14 $\quad\quad\quad\quad$ **if** $c \in \mathcal{C}_{N2}$ *in form* $L_1(X_1), L_2(X_2) \rightarrow \perp$ *and* $\exists h_1$ *such that* $h_1(L_i(X_i)) = L$ **then**
15 $\quad\quad\quad\quad\quad$ Let q' be $q'() \leftarrow h_1(L_{\bar{i}}(X_{\bar{i}}))$;
16 $\quad\quad\quad\quad\quad$ **if** $ans(q')$ *is positive* **then**
17 $\quad\quad\quad\quad\quad\quad valid = False$;
18 $\quad\quad\quad\quad$ **if** $c \in \mathcal{C}_K$ *is in form* $L_1(\boldsymbol{Y}, X_1, \boldsymbol{Z}_1), L_2(\boldsymbol{Y}, X_2, \boldsymbol{Z}_2) \rightarrow X_1 = X_2$ *and* $\exists h_1$ *such that* $h_1(L_i(\boldsymbol{Y}, X_i, \boldsymbol{Z}_i)) = L$ **then**
19 $\quad\quad\quad\quad\quad$ Let q' be $q'(X_{\bar{i}}) \leftarrow h_1(L_{\bar{i}}(\mathbf{Y}, X_{\bar{i}}, \mathbf{Z}_{\bar{i}}))$;
20 $\quad\quad\quad\quad\quad$ **if** $ans(q')$ *is not a singleton or* $ans(q') \neq \{h_1(X_i)\}$ **then**
21 $\quad\quad\quad\quad\quad\quad valid = False$;
22 $\quad\quad$ **if** $valid$ **then**
23 $\quad\quad\quad \mathcal{R} = \mathcal{R} \cup t$;
24 $\quad$ **return** $\mathcal{R}$;

bedded constraints allow the verification to be performed during query evaluation.

3.6.1 An Informal Description

Given a CQ q and a set of constraints $\mathcal{C}$, the rewriting step is done by using positive and negative constraints in $\mathcal{C}$ as rewriting rules. More precisely, if the body of a constraint c matches atoms in the body of the query q, then c is used as rewriting-rule to reformulate q into a new query that replaces q. By integrating constraint c to the query, the validation and the query evaluation are performed together, thereby avoiding validating extremely large number of intermediate results. Let us consider examples to illustrate the situations our query rewriting algorithm tackles with.

Example 3.3 Query q_1 below looks for professors who were born in a foreign country. Consider a user's context that is composed of c_{P_1}, c_{P_2} and c_{P_3} in the table 3.1.

$$q_1(X_1) \leftarrow professor(X_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$
$$c_{P_1} : professor(X) \rightarrow teacherOf(X, Y).$$
$$c_{P_2} : teacherOf(X, Y) \rightarrow offeredCourseAt(Y, Z).$$
$$c_{P_3} : professor(X) \rightarrow employeeGov(X).$$

In this context, we see $body(q_1)$ as a set of atoms capable of triggering constraints and producing new atoms that should be added to the query's body. This operation corresponds to a chase computation as mentioned in Section 3.3 , which starts with the atoms in $body(q_1)$. More precisely, atom $professor(X_1)$ in $body(q_1)$ and $body(c_{P_1})$ unify, and their MGU is $\{\sigma(X) = X_1\}$. This indicates that atom $professor$ in the $body(q_1)$ triggers the constraint c_{P_1}. Moreover, in this case, the MGU σ is an homomorphism from $body(c_{P1})$ to $body(q_1)$, thus the rewriting step is done by adding the $\sigma(head(c_{P_1}))$ into $body(q_1)$. Intuitively, answers of this new query not only are the answers of q_1 but also satisfy the constraint c_{P_1}. The result of this step is:

$$q'_1(X_1) \leftarrow professor(X_1), teacherOf(X_1, N_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$

Note that the existential variable Y in $head(c_{P_1})$ is replaced by a new fresh variable N_1 in $body(q'_1)$. Next, the new atom $teacherOf(X_1, N_1)$ in q'_1 continues triggering the constraint c_{P_2}, then q'_1 is rewritten into a new one by integrating $\sigma_2(head(c_{P_2}))$ to its body, and so on. The chase process terminates at the fix-point where there is no more new atom added into the body of the query. In this context, the new rewritten query, that the system should send to data providers, is:

$$q'_1(X_1) \leftarrow professor(X_1), teacherOf(X_1, N_1), offeredCourseAt(N_1, N_2), employeeGov(X_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$

Answers of new query q'_1 satisfy both q_1 and all the three above constraints, thus we can gain an equivalent results when evaluating q'_1 against databases instead of evaluating q_1 and then validating the constraints. □

Next, let's consider a more complex example with two kinds of constraints.

Example 3.4 Consider the query q_1 as in the above example but under a new context that is composed of c_{p1} and c_{N2}.

$$q_1(X_1) \leftarrow professor(X_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$
$$c_{P_1} : professor(X) \rightarrow teacherOf(X, Y).$$
$$c_{N_2} : teacherOf(X, Y), headOf(X, Z) \rightarrow \bot.$$

Similarly to Example 3.3, the positive constraint c_{P_1} is triggered by the atom $professor(X_1)$ in the query q_1, that results in the following rewritten query:

$$q'_1(X_1) \leftarrow professor(X_1), teacherOf(X_1, N_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$

Intuitively, the new atom $teacherOf(X_1, N_1)$ in q'_1 and the atom $teacherOf(X, Y)$ in c_{N2} unify, so c_{N2} is triggered. By using c_{N2} in this context, we want to obtain teachers who are not in the head of a department. In the case of datasets in Table 3.2, they are *Peter*, *Tom* and *Alice*. The solution for not dealing with negative atoms in the rewritten query is to store auxiliary complementary instances. The computation of these complementary queries is explained in Section 3.6.3. For our example, suppose *Peter*, *Tom* and *Alice* are stored in an auxiliary relation, called *aux_teacherOf*. Once we have this new relation, we replace the negative constraint c_{N2} with a positive constraint as follows:

$$c_{aux_1} : teacherOf(X, Y) \rightarrow aux_teacherOf(X, Y).$$

In this way, we treat the negative constraint c_{N2} in the same way as the positive one c_{P_1}. More particularly, q'_1 is rewritten as follows:

$$q'_1(X_1) \leftarrow professor(X_1), teacherOf(X_1, N_1), aux_teacherOf(X_1, N_1), placeOfBirth(X_1, Z_1), foreignCountry(Z_1).$$

This rewritten query is evaluated on the auxiliary database $\mathcal{S}_{aux}$ relation and the datasets $\overline{(\mathcal{S}, \tau)}$ as illustrated in Table 3.2. □

In some situations, the above rewriting technique should be revised, as illustrated by the following example.

Example 3.5 Consider query q_2, and constraint c_{p2} imposing restrictions on teachers who teach database - they should do research in the database domain:

$$q_2(X) \leftarrow teacherOf(X, Y).$$
$$c_{p2} : teacherOf(Z, DB) \rightarrow researchesIn(Z, DB).$$

Notice that no restriction is imposed on teachers in other domains. Here unlike Example 3.3, the unifier between predicates *teacherOf* of $body(q_2)$ and $body(c_{P2})$ is not a homomorphism from $body(c_{P2})$ to $body(q_2)$ (but the inverse). Moreover a query $q'_2(X) \leftarrow teacherOf(X, DB)$, $researchesIn(Z, DB)$ would ignore the teachers of all other domains, so we cannot apply the chase as in Example 3.3. In this case, our proposal is to replace q_2 by the union of the two following queries:

$$q_{2.1}(X) \leftarrow teacherOf(X, Y), Y \neq DB.$$
$$q_{2.2}(X) \leftarrow teacherOf(X, DB), researchesIn(X, DB).$$

The first query is created by adding a difference comparison formula of the constant

to deal with results that are not concerned by the constraint. While we add the corresponding equality and integrate the $head(c_{p2})$ into the given query to cover results related to the constraint, *i.e.* both the $body(c_{p2})$ and the $head(c_{p2})$ appear in the second query. □

This solution is explained with more details in Section 3.6.4.

The following section outlines the general process for rewriting queries. The method includes a preprocessing step where negative constraints are translated to constraints involving an auxiliary database.

3.6.2 General schema of the query rewriting process

Session 3.1 showed an overview of our query system, in which the module *Validator* is responsible for rewriting queries and validating answers with respect to the user's context. In this session, we explain in more details how it works. Figure 3.2 offers a closer look at this module.

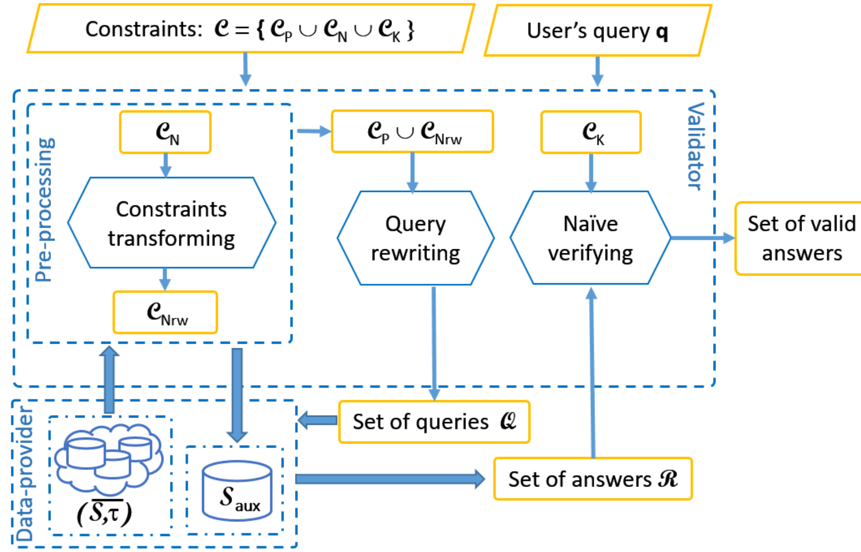

Figure 3.2 – Module Validator

The module *Validator* receives the user's query q and a set of quality constraints $\mathcal{C}$, which is constituted from three disjoint sets: positive constraints set $\mathcal{C}_P$, negative constraints set $\mathcal{C}_N$ and key constraints set $\mathcal{C}_K$. However, Validator treats each type in different ways. First, negative constraints $\mathcal{C}_N$ are transformed into a new set $\underline{\mathcal{C}_{Nrw}}$ in a preprocessing step. This translation needs access to the data in sources $(\mathcal{S}, \tau)$and may introduce new auxiliary relations. Auxiliary relations can be stored in an *auxiliary central database*, denoted by S_{aux}. However they can also be defined as views and evaluated over the database instance when needed. Then, these rewritten negative constraints together with positive constraints are used as rewriting rules to reformulate the initial query q. Basically, the reformulation process is an iterative

series of chase steps as the informal description in the above session. Thanks to preprocessing step, rewritten negative constraints can be treated in the same way as positive ones. The reformulation process results in a set of rewritten queries $\mathcal{Q}$, which then are evaluated against data on the auxiliary central database S_{aux} and sources $\overline{(\mathcal{S},\tau)}$. Now, let us denote by $\mathcal{R}$ the set of answers obtained from the evaluation of $\mathcal{Q}$. All these answers are valid with respect to $\mathcal{C}_P$ and $\mathcal{C}_N$ (see proofs in Section 3.6.5). The validation with respect to key constraints is done by the naive approach, *i.e.* each resulting tuple in $\mathcal{R}$ is verified by generating and evaluating subsidiary queries. Thus, at this (last) step non valid resulting tuples with respect to constraints in C_K are eliminated. As all tuples in $\mathcal{R}$ already satisfy constraints in C_P and C_N, the final result contains only valid answers.

3.6.3 Preprocessing Negative Constraints

The translation of negative constraints implies the definition of auxiliary instances to be stored in $\mathcal{S}_{aux}$. Algorithm 5 summarizes this preprocessing while the following definition introduces the queries used to compute auxiliary instances.

Definition 3.7 (Auxiliary Relations from Negative Constraints) Given a negative constraint $L_1(X,Y), L_2(X,Z), comp \rightarrow \bot$, where X, Y and Z are sequences of variable. We define auxiliary relations $L^c_{1,\overline{L_2}}$ and $L^c_{2,\overline{L_1}}$ as complementary relations of L_1 *w.r.t.* L_2 and, respectively, of L_2 *w.r.t.* L_1 the instances obtained by the evaluation of the following datalog programs on a given database source $\overline{(\mathcal{S},\tau)}$.

Computation of $L^c_{1,\overline{L_2}}$:	Computation of $L^c_{2,\overline{L_1}}$:
$q_1(X) \leftarrow L_1(X,Y)$	$q_1(X) \leftarrow L_2(X,Y)$
$q_2(X) \leftarrow L_2(X,Y), comp\|_{L_2}$	$q_2(X) \leftarrow L_1(X,Y), comp\|_{L_1}$
$q_3(X) \leftarrow q_1(X), \neg q_2(X)$	$q_3(X) \leftarrow q_1(X), \neg q_2(X)$
$L^c_{1,\overline{L_2}}(X,Y) \leftarrow L_1(X,Y), q_3(X)$	$L^c_{2,\overline{L_1}}(X,Y) \leftarrow L_2(X,Y), q_3(X)$

or, equivalently, the evaluation of the following relational algebra queries:
$L^c_{1,\overline{L_2}} := L_1 \bowtie (\Pi_X(L_1) \setminus \Pi_X(\sigma_{comp|L_2}(L_2))$ and
$L^c_{2,\overline{L_1}} := L_2 \bowtie (\Pi_X(L_2) \setminus \Pi_X(\sigma_{comp|L_1}(L_1))$ □

Let c_N be a negative constraint. As a result of the translation process over c_N we can have one of the following situations.

- When the body of c_N contains facts, but these facts do not exist in the database instance $\overline{(\mathcal{S},\tau)}$, the constraint c_N is not added into the set C_{Nrw}. Indeed, in this special case, the constraint forbids the existence of a specific fact. If this fact is not true in $\overline{(\mathcal{S},\tau)}$, the query can be evaluated "as it is" and no auxiliary relation is needed. In fact, c_N can be ignored for this database instance. For example, suppose the context has the two following constraints.

$$professor(X), (X = Tom) \rightarrow \bot.$$
$$teacherOf(X,Y), takesCourse(X,Y), (X = Bob), (Y = DB) \rightarrow \bot.$$

Assuming that tuples $professor(Tom)$, $teacherOf(Bob, DB)$ and $takesCourse(Bob, DB)$ do not exist in the database, Algorithm 5 tests on lines 8 and 15 fail and no new constraint is inserted in C_{Nrw}.

- When the body of c_N contains facts and these facts are true in the database instance $\overline{(\mathcal{S}, \tau)}$, the constraint c_N is translated into new constraints. For each fact f in $body(c_N)$ there is a new constraint of the format $f \rightarrow \bot$, as shown on lines 9,16,17 of Algorithm 5.

  Following our previous example, in case the tuples $professor(Tom)$, $teacherOf(Bob, DB)$ and $takesCourse(Bob, DB)$ are true in the database instance we will have the following constraints in C_{Nrw}:

$$professor(Tom) \rightarrow \bot.$$
$$teacherOf(Bob, DB) \rightarrow \bot.$$
$$takesCourse(Bob, DB) \rightarrow \bot.$$

- When the body of c_N is not composed by facts, a new constraint is added to C_{Nrw}. In the case c_N has only one atom in its body; c_N itself is added to C_{Nrw} (line 11). Otherwise, when c_N has two atoms in its body (*i.e.*, $L_1(u_1), L_2(u_2) \rightarrow \bot$); c_N is translated into two new constraints (lines 21, 24). Each new constraint involves an auxiliary complementary relation. Relation $L^c_{1,\overline{L_2}}(u_1)$ ($L^c_{2,\overline{L_1}}(u_1)$) contains the complement of L_1 *w.r.t.* L_2 (respectively, L_2 *w.r.t.* L_1) and the join and selection conditions appearing in c_N. Definition 3.7 establishes the queries that should be evaluated in order to compute instances of $L^c_{1,\overline{L_2}}(u_1)$ and $L^c_{2,\overline{L_1}}(u_1)$, to be stored in $\mathcal{S}_{aux}$. Notice that all instances in $\mathcal{S}_{aux}$ are computed on the basis of the current instance $\overline{(\mathcal{S}, \tau)}$.

In Algorithm 5, u'_1 (and u'_2) are extension of u_1 (res. u_2) where constants in u_1 are replaced with the variables in corresponding equalities (in *comp*). Function *ComputeAuxiliaryRelation* evaluates the queries established by Definition 3.7. The evaluation of such queries is only possible by a system dealing with negation. By introducing a preprocessing step where negative rules are rewritten, our approach allows the use of an auxiliary database management system. In this way, different strategies became possible: (i) when $\overline{(\mathcal{S}, \tau)}$ does not allow negation, it is possible to fetch needed information in $\overline{(\mathcal{S}, \tau)}$ and perform negation operations in the auxiliary database; (ii) when $\overline{(\mathcal{S}, \tau)}$ allows negation, queries can be evaluated on $\overline{(\mathcal{S}, \tau)}$ and results stored in $\mathcal{S}_{Aux}$. In both cases, view definitions can be stored in $\mathcal{S}_{Aux}$ and evaluated when needed instead of materializing $\mathcal{S}_{Aux}$.

It is worth noting that when a query triggers a constraint whose head is $\bot$, the rewriting process on this query terminates, and it is removed from the set of rewritten queries $\mathcal{Q}$ (Figure 3.2).

Example 3.6 Consider the dataset $\overline{(\mathcal{S}, \tau)}$ as Table 3.3 and the negative constraint:

$$c_{N3} : worksFor(X, Y, Z), bornIn(X, W), (Z = CNRS), (W = France) \rightarrow \bot$$

according to Definition 3.7, two auxiliary relations $aux_worksFor^{c_{N3}}$ and $aux_bornIn^{c_{N3}}$ are computed:

Algorithm 5: Preprocessing Negative Constraints

Input : • A set of negative constraints $\mathcal{C}_N$ on $\mathbb{G}$;
• An access to the database instance $\overline{(\mathcal{S},\tau)}$ over which the preprocessing should be performed.

Output: • A new set $\mathcal{C}_{Nrw}$ of positive constraints obtained from rewriting those in $\mathcal{C}_N$ over $\overline{(\mathcal{S},\tau)}$.
• The auxiliary database instance $\mathcal{S}_{Aux}$.

```
1  Function RewritingNegConstraints(C_N):
2      C_Nrw = ∅; S_aux = ∅
3      foreach constraint c ∈ C_N do
4          switch c according to its format do
5              case L(u) → ⊥ do
6                  if u has only constants then
7                      Evaluate the boolean query q_aux() ← L(u)
8                      if the result of q_aux() is true then
9                          Add L(u) → ⊥ to C_Nrw
10                 else
11                     Add L(u) → ⊥ to C_Nrw
12             case L1(u1), L2(u2) → ⊥ do
13                 if u1 and u2 have only constants then
14                     Evaluate boolean queries q_aux1() ← L1(u1) and
                        q_aux2() ← L2(u2) over instance (S, τ)‾
15                     if both results are true then
16                         Add L1(u1) → ⊥ to C_Nrw
17                         Add L2(u2) → ⊥ to C_Nrw
18                 else
                       /* In this case, according to Definition 3.3,
                          var(u1) ∩ var(u2) ≠ ∅.                          */
19                     L^c_{1,‾L2}(u'1) := ComputeAuxiliaryRelation(L1, L2, c)
20                     Add L^c_{1,‾L2}(u1) to S_Aux
21                     Add L1(u1) → L^c_{1,‾L2}(u1) to C_Nrw
22                     L^c_{2,‾L1}(u'2) := ComputeAuxiliaryRelation(L2, L1, c)
23                     Add L^c_{2,‾L1}(u2) to S_Aux
24                     Add L2(u2) → L^c_{2,‾L1}(u2) to C_Nrw
25     return C_Nrw and S_aux;
```

bornIn	worksFor	employeeGov
bornIn(Ann, France)	worksFor(Ann, LIFO, UnivOrleans)	employeeGov(Ann)
bornIn(Tom, USA)	worksFor(Tom, LIFO, UnivOrleans)	employeeGov(Bob)
bornIn(Bob, France)	worksFor(Bob, Paris, CNRS)	employeeGov(Alice)
bornIn(Alice, UK)	worksFor(Alice, Nice, CNRS)	
	worksFor(Peter, LIFAT, UnivTours)	

Table 3.3 – Datasets

aux_bornIn	aux_worksFor
$aux_bornIn^{c_{N3}}$(Ann, France)	$aux_worksFor^{c_{N3}}$(Tom, LIFO, UnivOrleans)
$aux_bornIn^{c_{N3}}$(Tom, USA)	$aux_worksFor^{c_{N3}}$(Alice, Nice, CNRS)
	$aux_worksFor^{c_{N3}}$(Peter, LIFAT, UnivTours)

Table 3.4 – Auxiliary Datasets

$q_{11}(X) \leftarrow worksFor(X, Y, Z)$
$q_{12}(X) \leftarrow bornIn(X, France)$
$q_{13}(X) \leftarrow q_{11}(X), \neg q_{12}(X)$
$aux_worksFor^{c_{N3}}(X, Y, Z) \leftarrow worksFor(X, Y, Z), q_{13}(X)$
$q_{21}(X) \leftarrow bornIn(X, W)$
$q_{22}(X) \leftarrow worksFor(X, Y, CNRS)$
$q_{23}(X) \leftarrow q_{21}(X), \neg q_{22}(X)$
$aux_bornIn^{c_{N3}}(X, W) \leftarrow bornIn(X, W), q_{23}(X)$
and they are then added into the auxiliary database S_{aux} as in Table 3.4. In this case, the negative constraint c_{N3} is transformed into two constraints as following:

$$worksFor(X, Y, CNRS) \rightarrow aux_worksFor^{c_{N3}}(X, Y, Z)$$
$$bornIn(X, France) \rightarrow aux_bornIn^{c_{N3}}(X, W)$$

By using these alternative constraints, tuples satisfying $worksFor(X, Y, CNRS)$ ($bornIn(X, France)$ respectively) must be tuples in the auxiliary relation $aux_worksFor$ (respect. aux_bornIn), whereby those tuples satisfy constraint c_{N3}.

Now, suppose that we have a query:

$$q(X) \leftarrow worksFor(X, Y, CNRS), employeeGov(X)$$

Intuitively, the atom $worksFor(X, Y, CNRS)$ in $body(q)$ triggers the constraint c_{N3}. Thus, the result of the rewriting step is:

$$q(X) \leftarrow worksFor(X, Y, CNRS), aux_worksFor^{c_{N3}}(X, Y, Z), employeeGov(X)$$

With the database instance of Table 3.3, the answer for q is *Alice*. □

3.6.4 Algorithm *RewriteQuery*

In this section, we present in detail the query rewriting algorithm in our approach.

Given a query q, and a set of constraints $\mathcal{C}_{rw}$, which consists of constraints in $\mathcal{C}_P$ and those resulted from Algorithm 5, the rewriting process consists of performing the chase algorithm, starting with the atoms in $body(q)$. The idea is similar to the one used in [32, 33] (see discussion in Section 2.4). Our rewriting algorithm denoted by *RewriteQuery* (Algorithm 7) proposes an iteration over constraints and queries. At each iteration, a step of the chase is performed by Algorithm 6 to incorporate constraints in $\mathcal{C}_{rw}$ to the body of q.

Let us consider in detail Algorithm 6. A given constraint c is integrated into the query q if it is triggered by an atom in q. An atom $L(u) \in body(q)$ triggers a constraint c if there exists a *mgu* between $L(u)$ and $body(c)$. As after translating $\mathcal{C}_N$ into $\mathcal{C}_{Nrw}$ constraints have only one atom in their bodies, testing whether the constraint c is triggered by an atom L in $body(q)$ is to check the existence of a *mgu* θ between $L(u)$ and $body(c)$ (Algorithm 6, line 3). Moreover, θ should ensure satisfiability, as illustrated in the example below.

Example 3.7 Consider the query and the constraint as follows:

$$q(X) \leftarrow A(X, Y, Z), Y = a, Z = b$$
$$c : A(X_1, X_1, X_1) \rightarrow B(X_1, Y_1)$$

Clearly θ such that $\theta(X) = X_1, \theta(Y) = X_1, \theta(Z) = X_1$ is a mgu of $body(c)$ and the atom A in $body(q)$. However, $\theta(q)$ leads to $a = b$, two different constants, and thus it is not satisfiable. In this case c is not triggered by q. □

Now, once the condition on line 4 is satisfied, the constraint c should be matched to atom $L(u)$. To this end, variables in c are renamed by variables in q by using the *one-way mgu* defined below.

Definition 3.8 (One-way MGU) Let l_1 and l_2 two literals which are unifiable. A one-way unifier denoted by $\theta_{l_1 \rightarrow l_2}$ is a *mgu* for l_1 and l_2 such that all variables of $\theta_{l_1 \rightarrow l_2}(l1)$ are variables of $l2$. □

For example, let $l_1 = A(X_1, a, b, Y_1, Y_1)$ and $l_2 = A(a, X_2, Y_2, Z_2, U_2)$ which are unifiable. A possibility for the one-way mgu from l_1 to l_2 is $\{\theta_{l_1 \rightarrow l_2}(X_1) = a, \theta_{l_1 \rightarrow l_2}(Y_1) = Z_2, \theta_{l_1 \rightarrow l_2}(X_2) = a, \theta_{l_1 \rightarrow l_2}(Y_2) = b, \theta_{l_1 \rightarrow l_2}(U_2) = Z_2\}$ and $\{\theta_{l_2 \rightarrow l_1}(X_1) = a, \theta_{l_2 \rightarrow l_1}(X_2) = a, \theta_{l_2 \rightarrow l_1}(Y_2) = b, \theta_{l_2 \rightarrow l_1}(Z_2) = Y_1, \theta_{l_2 \rightarrow l_1}(U_2) = Y_1\}$ for the one-way mgu from l_2 to l_1. Clearly $\theta_{l_1 \rightarrow l_2}(l1) = \theta_{l_1 \rightarrow l_2}(l2)$.

The need of the *one-way mgu* is illustrated by the example below.

Example 3.8 Let $c : A(X_1, a, b, Y_1, Y_1) \rightarrow B(X_1, Y_1)$ be a constraint and let $q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2)$ be a query.

Clearly, $l_1 = A(X_1, a, b, Y_1, Y_1)$ and $l_2 = A(a, X_2, Y_2, Z_2, U_2)$ are unifiable but there is neither a homomorphism from the l_1 (*i.e.*, c's body) to l_2 nor a homomorphism from l_2 to l_1. However, intuitively, and since *mgu* θ exists, it is clear that it

is possible to find an instantiation of $A(a, X_2, Y_2, Z_2, U_2)$ that triggers c. In other terms, an answer for q can trigger c imposing the constraint verification.

An one-way unifier, such as $\{\theta_{l_1 \to l_2}(X_1) = a, \theta_{l_1 \to l_2}(Y_1) = Z_2, \theta_{l_1 \to l_2}(X_2) = a, \theta_{l_1 \to l_2}(Y_2) = b, \theta_{l_1 \to l_2}(U_2) = Z_2\}$ applied to c gives $A(a, a, b, Z_2, Z_2) \to B(a, Z_2)$. Now it is possible to find a homomorphism from l_2 to l_1.

Notice that an one-way unifier allows Algorithm 6 to produce new queries q_1 and q_2, according to lines 12-19.
Indeed, continuing our example, let us consider the homomorphism h from l_2 to $\theta_{l_1 \to l_2}(l_1)$ such that: $h(X_2) = a$, $h(Y_2) = b$, $h(Z_2) = Z_2$, $h(U_2) = Z_2$

Then, according to Algorithm 6 (lines 12-19) we obtain the following queries :

$q_2 :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), B(a, Z_2), X_2 = a, Y_2 = b, U_2 = Z_2$

$q_{1_1} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), X_2 \neq a,$

$q_{1_2} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), Y_2 \neq b$

$q_{1_3} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), U_2 \neq Z_2$

The one-way unifier allows rewriting a query on the basis of the unification of only two atoms - one in the query's body and the one in the constraint's body. Without it, it would be necessary to apply a mgu to the whole query's body at each step. Let us illustrate the problem by supposing that we do not use an one-way unifier but just a mgu, as for example $\{\theta(X_1) = a, \theta(X_2) = a, \theta(Y_2) = b, \theta(Z_2) = Y_1, \theta(U_2) = Y_1\}$.

Clearly, in this case, $\theta(c)$ gives $A(a, a, b, Y_1, Y_1) \to B(a, Y_1)$.

In this situation it is still possible to have a homomorphism h from $l_2 = A(a, X_2, Y_2, Z_2, U_2)$ to $l_1 = A(a, a, b, Y_1, Y_1)$: $\{h(X_2) = a, h(Y_2) = b, h(Z_2) = Y_1, h(U_2) = Y_1\}$. Now, according to Algorithm 6 (lines 12-19), we obtain the following queries :

$q'_2 :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), B(a, Y_1), X_2 = a, Y_2 = b, Z_2 = Y_1, U_2 = Y_1$

$q'_{1_1} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), X_2 \neq a$

$q'_{1_2} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), Y_2 \neq b$

$q'_{1_3} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), Z_2 \neq Y_1$

$q'_{1_4} :\ q(X_2, Y_2, Z_2, U_2) \leftarrow A(a, X_2, Y_2, Z_2, U_2), C(U_2), U_2 \neq Y_1$

Compared with the previous result, we observe that queries q_2 and q'_2 are identical. However, there are differences between queries q_1 and q'_1. Instead of query q_{1_3} we have queries q'_{1_3} and q'_{1_4} which are not well formed. Comparison atoms in their bodies contain variables which are not bounded by a relation. □

Algorithm 6: A step of chase processing

Input : A conjunctive query $q : \tau_{in}$ and a constraint $c \in \mathcal{C}_{rw}$ where $\mathcal{C}_{rw} = \mathcal{C}_P \cup \mathcal{C}_{Nrw}$

Output: A set Q of rewritten queries $q_r : \tau_{in}$, such that for each $q_r : \tau_{in}$ we have $head(q_r) = head(q)$.

1 **Function** `StepChase`($q : \tau_{in}$, c):
2 **foreach** *atom* $L(u) \in body(q)$ *such that:*
3 *there is a mgu* $\theta : \theta(L(u)) = \theta(body(c))$ *for which*
4 $\theta(body(q))$ *is satisfiable* **do**
5 Denote query $q : \tau_{in}$ by $q(u_0) \leftarrow \beta(u_1), L(u)$;
6 Let σ_c be $\theta_{body(c) \rightarrow L(u)}$ (Definition 3.8) ;
7 **if** $\exists$ *homomorphism* ν *from* $body(\sigma_c(c))$ *to* $L(u)$ **then**
8 **if** $\neg Isomorphic(\nu(\sigma_c((head(c))), q)$ **then**
9 $q(u_0) \leftarrow \beta(u_1), L(u), (\nu(\sigma_c((head(c)))$;
10 $Q := Q \cup \{q : \tau_{in}\}$;
11 **else**
12 Let h be a homomorphism such that $h(L(u)) = \sigma_c(body(c))$;
13 Let $Q := \emptyset$;
14 Let $q_2 : \tau_{in}$ be $q(u_0) \leftarrow \beta(u_1), L(u), h(\sigma_c(head(c)))$;
15 **foreach** *variable* x *in* u *for which* $h(x) = a$ *and* $h(x) \neq x$, *where* a *is a term* **do**
16 Let $q_1 : \tau_{in}$ be $q(u_0) \leftarrow \beta(u_1), L(u), (x \neq a)$;
17 $Q := Q \cup \{q_1 : \tau_{in}\}$;
18 $body(q_2) := body(q_2) \wedge (x = a)$;
19 $Q := Q \cup \{q_2 : \tau_{in}\}$;
20 return(Q);

The notion of *one-way* unifier is essential when constraints have constants or repeated variables in their bodies. Algorithm 6, line 12, finds the homomorphism h once the one-way unifier is applied to the body of c.

Now, let us come back to the general idea of Algorithm 6. Once θ and $\theta_{body(c)\to L(u)}$ are determined on line 6, we test whether there is a homomorphism ν from $body(\sigma_c(c))$ to $L(u)$. This is the criteria to decide *how* c is integrated into q (line 7). Indeed, we have two main situations:

1. When the homomorphism ν exists, the query's body is completed with the atom computed from the head of $\sigma_c(c)$ (line 9). Notice that the new atom is added into $body(q)$ if it is not isomorphic to any atom already existing in q. It is done by function $Isomorphism$ at line 8 that verifies the isomorphism between every atom of $body(q)$ and the new atom being considered adding to q. In this way we avoid triggering one constraint many times by an atom in q, and adding redundant atoms into the body of the query. Due to this condition, the result of the rewriting step may be empty, *i.e.* at a given iteration c is not integrated into q (since its consequence is already integrated in q).

   We recall that our constraints respect some syntactic restrictions, namely:

   - The set $\mathcal{C}_P$ is a set of weakly acyclic TGD ([35]). Roughly, a set of TGD is acyclic if it does not allow for cascading of labelled null creation during the chase.
   - Constraints resulting from the translation of negative constraints (Algorithm 5) cannot trigger other constraints. Indeed their heads are on new auxiliary predicates that do not appear in the initial global schema.

2. When the homomorphism ν does not exist, the reason is summarized by one of the following situations: (i) we cannot map a constant in $body(c)$ to a variable or a different constant in $L(u)$ or (ii) we cannot map a variable in $body(c)$ (appearing more than once) to different variables in $L(u)$

   Indeed, in Example 3.5, no homomorphism from $body(\sigma_c(c_{p2}))$ to $L = teacherOf(X, Y)$ is possible. However, as there is a *mgu* between L and $body(c)$, there is a homomorphism h from L to $body(\sigma_c(c_{p2}))$ (line 12). In this case, we generate two queries, namely:

   - a query (q_2 in the Algorithm 6) that deals with results (*i.e.*, possible query instantiations) involving constraint c, and
   - a set of queries (q_1 in the *foreach* loop at line 15-18) dealing with results that are *not* concerned by c.

   Indeed, in the above cases, c can be written by using comparison atoms (*comp*) containing equalities. During the rewriting process, Algorithm 6 generates queries q_1 by adding into $body(q)$ the negation of these comparison atoms (*i.e.*, $\neg comp$).

> For instance, in our Example 3.5, $q_{2.1}$ selects people who do not teach *DB*. With the database instance of Table 3.2, the answer for $q_{2.1}$ is $Tom, Alice, Peter$. Query $q_{2.2}$ deals with results *concerned* by the constraint. It selects two kinds of people: (i) those who are database researchers and *only* teach *DB* and (ii) those who teach and do research in the database domain but also teach other subjects. In this example, the answers for q_2 on the instance of Table 3.2 are *Bob*, *Tom*, *Alice* and *Peter*. Notice that *Bob* is not an answer for $q_{2.1}$, but it is the answer to $q_{2.2}$. The result of q_2 is the union of the answers for $q_{2.1}$ and $q_{2.2}$.

Note that rewritten queries dealing with results not concerned by c (*i.e.* q_1 in Algorithm 6) introduce negative *comp* atoms. In the relational algebra they can be translated as a negative condition for the selection operation.

Let's now consider in detail Algorithm 7. The general idea here is to apply exhaustively the chase step with constraints in C_{rw} on q. More precisely, applying the chase step on q and a constraint $c \in C_{rw}$ results in a set of rewritten queries Q' (line 8) which may replace q in the next step. Recall that the purpose of integrating constraint c into q by applying the chase step on q and c is to filter answers of q that are valid with respect to c.

Algorithm 7: Query Rewriting

Input : A conjunctive query $q : \tau_{in}$ and a set of constraints $\mathcal{C}_{rw} = \mathcal{C}_P \cup \mathcal{C}_{Nrw}$

1 **Function** `RewriteQuery`($q : \tau_{in}$, $\mathcal{C}$):
2   $Q = \{q : \tau_{in}\}$;
3   **repeat**
4     $Changed := false$;
5   **foreach** $c \in \mathcal{C}$ **do**
6   **foreach** $q : \tau_{in} \in Q$ **do**
7         $replace := false$;
8   Q' := `StepChase`($q : \tau_{in}$, c);
          /* Algorithm 6 */
9   **foreach** $q' : \tau_{in} \in Q'$ **do**
10   **if** *($q' : \tau_{in}$ is more restricted than $q : \tau_{in}$) and ($q' : \tau_{in}$ is not contradictory)* **then**
11             $Q := Q \cup \{q' : \tau_{in}\}$;
12             $Changed := true$;
13             $replace := true$;
14   **if** *replace* **then**
15           $Q := Q \setminus \{q : \tau_{in}\}$;
16   **until** *not Changed*;
17   **return** Q;

Repeat (line 3)	foreach c in C_{rw} (line 5)	Q (beginning of one repeat iteration)	foreach q in Q	$Q' =$ *stepChase* (q, c)	q' not contradictory and more restricted than q	Q (end of one repeat iteration)	*changed*
1	c_1	$\{q\}$	q	$\{q_1\}$	$\{q_1\}$	$\{q_1\}$	T
	c_2	$\{q_1\}$	q_1	$\{q_{21}, q_{22}\}$	$\{q_{21}, q_{22}\}$	$\{q_{21}, q_{22}\}$	T
2	c_1	$\{q_{21}, q_{22}\}$	q_{21} q_{22}	$\{\ \}$ $\{q_{31}\}$	$\{\ \}$ $\{q_{31}\}$	$\{q_{21}, q_{22}\}$ $\{q_{21}, q_{31}\}$	F T
	c_2	$\{q_{21}, q_{31}\}$	q_{21} q_{31}	$\{q_{41}, q_{42}\}$ $\{\ \}$	$\{\ \}$ $\{\ \}$	$\{q_{21}, q_{31}\}$ $\{q_{21}, q_{31}\}$	F F

Table 3.5 – Intermediate results of Algorithm 7

Query q is replaced by Q' only if Q' is more restricted than q, *i.e.* $ans(Q')$ is included in $ans(q)$ (line 10). On the other hand, in Algorithm 6, for a given query q, each rewritten query q' is created by instantiating atoms or adding more atoms into $body(q)$. Thus, an answer to each q' is either equal to or included in the answer to q. When q' is equivalent to q, we can remove it from Q'. In other words, only rewritten queries (strictly) more restricted than q are considered replacing q in Q (lines 9-15).

The new set Q, in turn, will be chased with every constraint $c \in C_{Nrw}$ including constraints already considered in previous steps until no more new query is generated (*i.e.* there is no change in Q).

Example 3.9 Let's consider query q and two constraints c_1, c_2 as follows:

$$q(X) \leftarrow L_1(X, Y)$$
$$c_1 : L_1(X, Y) \rightarrow L_2(X, Y)$$
$$c_2 : L_2(X, a) \rightarrow L_1(X, b)$$

Table 3.5 illustrates the execution of Algorithm 6, step by step. Columns in this table indicate the moment of algorithm's execution being considered. The sixth column shows q' in $Q' = stepChase(q, c)$ such that q' is satisfiable and more restricted than q. The list of intermediate rewritten queries is in Table 3.6.

We can see that q'_{21}, q'_{22} and q'_{31} are removed from Q' by the *isomorphic condition* in Algorithm 6 (line 8), *i.e.* there exists an atom in $body(q)$ that is isomorphic to the added atom in the rewriting step. Queries q_{41} and q_{42} are removed by the condition at line 10 in Algorithm 7.
Thus, the rewriting result of q with respect to c_1 and c_2 is the set of two following rewritten queries:

$$q_{21}(X) \leftarrow L_1(X, Y), L_2(X, Y), Y \neq a$$
$$q_{31}(X) \leftarrow L_1(X, a), L_1(X, b), L_2(X, a), L_2(X, b)$$

Algo. 6	q'	Remark
	$q(X) \leftarrow L_1(X,Y)$	
(q, c_1)	$q_1(X) \leftarrow L_1(X,Y), L_2(X,Y)$	
(q_1, c_2)	$q_{21}(X) \leftarrow L_1(X,Y), L_2(X,Y), Y \neq a$ $q_{22}(X) \leftarrow L_1(X,a), L_2(X,a), L_1(X,b)$	
(q_{21}, c_1)	$q'_{21}(X) \leftarrow L_1(X,Y), L_2(X,Y), L_2(X,Y)$	isomophism
(q_{22}, c_1)	$q'_{22}(X) \leftarrow L_1(X,a), L_2(X,a), L_2(X,a), L_1(X,b)$	isomophism
(q_{22}, c_1)	$q_{31}(X) \leftarrow L_1(X,a), L_2(X,a), L_1(X,b), L_2(X,b)$	
(q_{21}, c_2)	$q_{41}(X) \leftarrow L_1(X,Y), L_2(X,Y), Y \neq a, Y \neq a$ $q_{42}(X) \leftarrow L_1(X,Y), L_2(X,Y), Y = a, L_1(X,b), Y \neq a$	not more restricted than q_{21} contradictory
(q_{31}, c_2)	$q'_{31}(X) \leftarrow L_1(X,a), L_2(X,a), L_1(X,b), L_1(X,b), L_2(X,b)$	isomophism

Table 3.6 – Intermediate rewritten queries

3.6.5 Correctness of Validation through rewriting queries

We now establish the correctness of Algorithm 7. We first consider some lemmas. The first lemma deals with the situation where $\mathcal{C}$ has only a positive constraint whose body cannot be mapped to an atom the query's body (although a unifier exits) (as on line 12 Algorithm 6). The corresponding rewritten query q_r computes all and only the q's answers which are valid with respect to $\mathcal{C}$.

Lemma 3.1 Let

$$q(u_0) \leftarrow \beta(u_1), L(u_2)$$

be a conjunctive query where u_0, u_1, u_2 are free tuples, u_0 contains only variables appearing in $body(q)$, $L(u_2)$ is an atom and $\beta(u_1)$ is the conjunction of the other atoms in the query's body.

Let c be a positive constraint for which the following conditions hold:

1. there exists a *mgu* capable of unifying $body(c)$ with $L(u_2)$. We note σ_c the substitution $\theta_{body(c) \rightarrow L(u_2)}$ from Definition 3.8,
2. there is no homomorphism ν such that $\nu(body(c)) = L(u_2)$.

Let h be a homomorphism such that $h(L(u_2)) = \sigma_c(body(c))$. Let q_r be a rewritten version of q, on the basis of c, denoted by the queries:

1. **(Q1)** $q_r(u_0) \leftarrow \beta(u_1), L(u_2), (x \neq a)$
   for each variable x in u_2 for which $h(x) = a$ and $h(x) \neq x$, where a is a term.
2. **(Q2)** $q_r(u_0) \leftarrow \beta(u_1), L(u_2), (x_1 = a_1), \ldots, (x_n = a_n), h(\sigma_c(head(c)))$
   where, for $i \in [1, n]$, x_i is a variable in u_2 and each a_i is a term such that $h(x_i) = a_i$ and $a_i \neq x_i$.

For any database instance $\overline{(\mathcal{S}, \tau)}$, we have

$$q(\overline{(\mathcal{S}, \tau)}, c) = q_r(\overline{(\mathcal{S}, \tau)})$$

where $q(\overline{(\mathcal{S},\tau)},c)$ is the set of *valid* answers of q (with respect to c) on $\overline{(\mathcal{S},\tau)}$ and $q_r(\overline{(\mathcal{S},\tau)})$ is the set of answers of q_r on $\overline{(\mathcal{S},\tau)}$. □

PROOF:
Part 1: We first prove that for any tuple t, if t is a valid answer for q with respect to constraint c then it will be an answer for q_r.

We start by considering two different situations

A) In query q, there exist variables in the free tuple u_0 which are also in u_2. More precisely, let us suppose that u_0 is composed of two other tuples. We write $u_0 = u, u_{0.1}$. Similarly we have $u_2 = u, u_{2.1}$. Therefore we can write q as $q(u, u_{0.1}) \leftarrow \beta(u_1), L(u, u_{2.1})$. Moreover, supposing that $t = t_1, t_2$ the instantiation format of q, obtained by $h_t(q)$ is $q(t_1, t_2) \leftarrow \beta(\alpha_1), L(t_1, \alpha_2)$, where t_2, α_1 and α_2 are also tuples of constants. Without loss of generality, we assume that t_1 is a unary tuple (therefore u is just a variable x).

We know that there is h such that $h(L(u_2)) = \sigma_c(body(c))$. The following situations should be considered

1. $h(x)$ is a constant.

   (a) When $h_t(x) = h(x)$, the instantiated atom $h_t(L(u_2))$ triggers c *i.e.*, $h_t(L(u_2)) = h_t(\sigma_c(body(c)))$. In such a case, as t is a valid answer we know that constraint c is triggered and the query $q_{aux}() \leftarrow h_t(\sigma_c(head(c))$ has a non-empty answer.

   Clearly, in this case, the query (Q2) above can be instantiated by h_t, giving:
   $$q_r(t_1, t_2) \leftarrow \beta(\alpha_1), L(t_1, \alpha_2), h_t(\sigma_c(head(c)))$$
   Indeed, if a is the constant such that $t_1 = (a)$, then we have $h_t(x) = a$, and the equalities appearing in the body of the query (Q2) are satisfied.

   In the instantiation of this rewritten query, the first two atoms are the same as those in the instantiation of q and the last one corresponds to the body of query q_{aux}. Thus, as the answer set of q_{aux} is not empty, this last atom of q_r will be instantiated with the same answer set, giving tuple t as an answer.

   REMARK: When t_1 is a n-ary tuple, the only difference is that there will be n conditions in the body of query (Q2), imposing for each variable x in u to be instantiated by the constant $h(x)$, which, in this case, equals $h_t(x)$. Thus, all these conditions are evaluated to true. The same argument follows.

   Clearly, the query (Q1) produces no answer. As $h_t(x) = a$, condition $(x \neq a)$ in the body of query (Q1) cannot be satisfied.

   REMARK: When t_1 is a n-ary tuple, the only difference is that there will be n queries similar to query (Q1). Each query will impose a condition of the form $(x \neq h(x))$ which is not satisfied since in this case, $h(x)$

equals $h_t(x)$ Thus, no query following the model of query (Q1) will produce a result.

(b) When $h_t(x) \neq h(x)$, the instantiated atom $h_t(L(u_2))$ does not trigger c. Let the instantiation of q be $q(t_1, t_2) \leftarrow \beta(\alpha_1), L(t_1, \alpha_2)$ to produce $t = (t_1, t_2)$ as a valid answer - since no constraint is triggered. In this case if $t_1 = (b)$, we have $h_t(x) = b$. Let $h(x) = a$.

The instantiation of query (Q2) by h_t results in a condition such as $b = a$ which is evaluated to false. Thus the evaluation of query (Q2) gives an empty set. However, the instantiation of query (Q1) by h_t gives $q(b, t_2) \leftarrow \beta(\alpha_1), L(b, \alpha_2), (b \neq a)$ which correspond to $h_t(q)$ for $t_1 = (b)$ as supposed above.

2. $h(x)$ is a variable. In this case, condition in the body of queries (Q1) and (Q2) are of the form $x = y$ or $x \neq y$ for a variable y appearing in u_2. Therefore, that is at least two variables in u_2 (denoted here by x and y) whose instantiations should be compared in terms of equality. Indeed, in such a situation there exist as least two variables that are the same in $body(c)$ (this situation can be characterized since we know that there is no homomorphism ν mapping $body(c)$ to $L(u_2)$ and $h(x)$ is a variable).

(a) When $h_t(x) = h_t(y)$, the instantiated atom $h_t(L(u_2))$ triggers c : as t is a valid answer we know that constraint c is triggered and the query $q_{aux}() \leftarrow h_t(\sigma_c(head(c))$ has a non-empty answer.

In this case, the query (Q2) can be instantiated by h_t, giving:

$$q_r(t_1, t_2) \leftarrow \beta(\alpha_1), L(t_1, \alpha_2), h_t(\sigma_c(head(c)))$$

Indeed, if a is the constant such that $t_1 = (a)$, then we have $h_t(x) = a$, $h(x) = y$ and $h_t(y) = a$. The equalities appearing in the body of the query (Q2) are satisfied.

In the instantiation of this rewritten query, the first two atoms are the same as those in the instantiation of q and the last one corresponds to the body of query q_{aux}. Thus, as the answer set of q_{aux} is not empty, this last atom of q_r will be instantiated with the same answer set, giving tuple t as an answer.

Similarly to case 1a query (Q1) produces no answer. As $h_t(x) = h_t(y) = a$, condition $(x \neq y)$ in the body of query (Q1) cannot be satisfied.

(b) When $h_t(x) \neq h_t(y)$, the instantiated atom $h_t(L(u_2))$ does not trigger c and the argument follows those developed in steps 1b and 2a.

B) In query q, no variable appearing in the free tuple u_0 appears in u_2. As in part A), validity is verified by considering two situations: (i) when the instantiation $h_t(L(u_2))$ triggers c, the result of query $q_{aux}() \leftarrow h_t(\sigma_c(head(c))$ should be non-empty, otherwise (ii) constraint c is not triggered and answer $h_t(u_0)$ is valid.

To prove that the evaluation of q_r gives all valid tuples resulting from the evaluation of q, we proceed as in part A). The proof is strictly the same - just the instantiation format of queries changes.

Part 2: We now prove that for any tuple t, if t is an answer for q_r , then t is a valid answer for q with respect to constraint c.

Let h_t be a homomorphism which instantiated q_r on $\overline{(\mathcal{S},\tau)}$ giving t as an answer. For a given h_t we know that, either (Q1) or (Q2) can be instantiated, but not both, since conditions in the body of these two rules are disjoint.

- Suppose there is an instantiation h_t for query (Q1) for which we have t as an answer.

  Clearly the conjunction $h_t(\beta(u_1), L(u_2))$ is also a possible evaluation for q which does not trigger c (since in this situation $h_t(x) \neq h(x)$ for each variable in u_2 respecting the properties established by this lemma). Thus, t is a valid answer with respect to c for q on $\overline{(\mathcal{S},\tau)}$.

- Suppose there is an instantiation h_t for (Q2) for which we have t as an answer.

  The conjunction $h_t(\beta(u_1), L(u_2))$ is also a possible evaluation for q which triggers c (here $h_t(x) = h(x)$ for each variable in u_2 respecting the properties established by this lemma). In this case, the fact $h_t(L(u_2))$ triggers c - only if there is a non-empty answer for query $q_{aux}() \leftarrow h_t(\sigma_c(head(c))$, tuple t is valid. As h_t on (Q2) gives an instantiation of $\sigma_c(head(c))$ on instance $\overline{(\mathcal{S},\tau)}$, we know that a non-empty answer exist. Thus t is valid with respect to c. □

The second lemma shows that when $\mathcal{C}$ has only one negative constraint of the form $c : A(v_1), B(v_2) \rightarrow \perp$ whose body is triggered by an atom in the query's body, the corresponding rewritten query q_r computes all and only the q's answers which are valid with respect to $\mathcal{C}$. Moreover, in this situation, query q_r involves auxiliary instances computed according to Definition 3.7 and stored in S_{aux}.

Lemma 3.2 Let

$$q(u_0) \leftarrow \beta(u_1), L(u_2)$$

be a conjunctive query where u_0, u_1, u_2 are free tuples, u_0 contains only variables appearing in $body(q)$, $L(u_2)$ is an atom and $\beta(u_1)$ is the conjunction of the other atoms in the query's body.

Let $c : A(v_1), B(v_2) \rightarrow \perp$ be a negative constraint where there is at least one common variable between v_1 and v_2, and for which there exists a *mgu* θ capable of unifying $L(u_2)$ with an atom in $body(c)$. Let $A(v_1)$ be this atom (that is A and L are the same predicate symbol). We denote by σ_c the substitution $\theta_{body(c)\rightarrow L(u_2)}$ from Definition 3.8. Let c_1 be the translation of c according to Algorithm 5 on database instance $\overline{(\mathcal{S},\tau)}$, *i.e.* $c_1 : A(v_1) \rightarrow A^{c}_{\overline{B}}(v_1)$ where $A^{c}_{\overline{B}}(u_2)$ is in S_{aux}.

Consider two different situations:

1. There is a homomorphism ν allowing a mapping of atom $\sigma_c(A(v_1))$ to $L(u_2)$. Let q_r be the rewritten version of q, on the basis of c_1, denoted by: **(Q0)** $q_r(u_0) \leftarrow \beta(u_1), L(u_2), \nu(A^{c}_{\overline{B}}(v_1))$.

2. There is no homomorphism ν such that $\sigma_c(A(v_1)) = L(u_2)$. Let h be a homomorphism such that $h(L(u_2)) = \sigma_c(A(v_1))$. Let q_r be a rewritten version of q, on the basis of c_1, denoted by the queries:

   (a) **(Q1)** $q_r(u_0) \leftarrow \beta(u_1), L(u_2), (x \neq a)$
   for each variable x in u_2 for which $h(x) = a$ and $h(x) \neq x$, where a is a term.

   (b) **(Q2)** $q_r(u_0) \leftarrow \beta(u_1), L(u_2), (x_1 = a_1), \ldots, (x_n = a_n), h(\sigma_c(A^{c}_{\overline{B}}(v_1)))$
   where, for $i \in [1, n]$, x_i is a variable in u_2 and each a_i is a term such that $h(x_i) = a_i$ and $a_i \neq x_i$.

For any database instance $\overline{(\mathcal{S}, \tau)}$, we have

$$q(\overline{(\mathcal{S}, \tau)}, c) = q_r(\overline{(\mathcal{S}, \tau)} \cup S_{aux})$$

where $q(\overline{(\mathcal{S}, \tau)}, c)$ is the set of *valid* answers of q (with respect to c) on $\overline{(\mathcal{S}, \tau)}$ and $q_r(\overline{(\mathcal{S}, \tau)})$ is the set of answers of q_r on $\overline{(\mathcal{S}, \tau)} \cup S_{aux}$. □

PROOF:
For situation 1, query q is rewritten into just one query **(Q0)**. Let $t \in q(\overline{(\mathcal{S}, \tau)}, c)$. Thus, as t is a valid answer with respect to c, we know that there exists a homomorphism h_t such that $h_t(body(q)) \subseteq \overline{(\mathcal{S}, \tau)}$ and $q_{aux}() \leftarrow h_t(\nu(B(v_2)))$ is $false$ (has no solution) in $\overline{(\mathcal{S}, \tau)}$. From Definition 3.7, of $A^{c}_{\overline{B}}$ we deduce that $q'_{aux}() \leftarrow h_t(\nu(A^{c}_{\overline{B}}(v_1)))$ has *true* as solution in S_{aux}. Then t is also a solution of q_r on $\overline{(\mathcal{S}, \tau)} \cup S_{aux}$.

Conversely, let t be a solution for $q_r(\overline{(\mathcal{S}, \tau)} \cup S_{aux})$. In this situation, there exists a homomorphism h_t such that $h_t(body(q_r)) \subseteq \overline{(\mathcal{S}, \tau)} \cup S_{aux}$. More precisely, we know that $h_t(\beta(u_1), L(u_2)) \subseteq \overline{(\mathcal{S}, \tau)} \cup S_{aux}$ and $h_t(\nu(A^{c}_{\overline{B}}(v_1)) \subseteq S_{aux}$. So the solution for query $q'_{aux}() \leftarrow h_t(\nu(A^{c}_{\overline{B}}(v_1)))$ is *true*. Now, following Definition 3.7, we know that the computation of facts in $A^{c}_{\overline{B}}$ is performed over instance $\overline{(\mathcal{S}, \tau)}$, on the basis of facts in A and not in B. In this way we deduce that $q'_{aux}() \leftarrow h_t(\nu(B(v_2)))$ is $false$ (has no solution) in $\overline{(\mathcal{S}, \tau)}$. In conclusion, t is a solution of q in $\overline{(\mathcal{S}, \tau)}$ and t is valid with respect to c. Clearly, t is in $q(\overline{(\mathcal{S}, \tau)}, c)$.

For situation 2, the proof follows the lines of the proof of Lemma 3.1 by using the same arguments shown above, *i.e.*, by using a negative constraint c and its translation c_1, according to Algorithm 5, and by considering the auxiliary database S_{aux}, whose instances are computed by following Definition 3.7. □

We are now ready to establish the termination together with the soundness and completeness of our algorithm.

Proposition 1 Given a conjunctive query q and a set of constraints $\mathcal{C}_{rw}$, Algorithm 7 finishes in a finite number of steps. □

PROOF :
To prove the termination of our rewriting algorithm we just need to observe that it is composed of two parts. The first part deals with situations where the homomorphism ν exists. Therefore, when our rewriting algorithm is restricted to these situations (*i.e.*, when only lines 7 to 9 are executed in function StepChase (Algorithm 6)), it corresponds to the chase with non weakly acyclic constraints $\mathcal{C}_1$ (tgd) of a conjunctive query q (and therefore to the chase phase of algorithms introduced in [33]). Thus, as in [33], which is based on proofs in [35], it terminates. The second part considers the situation where the homomorphism ν does not exist. In this situation, a query q generates at least two other queries. Clearly, the number of new queries that can be created during the execution of lines 12 to 19 is finite and bounded by the number of variables for which conditions on line 15 hold. From these observations and since the number of constraints and queries is finite, Algorithm 7 terminates. Moreover, after each execution of Algorithm 6 the new generated queries are compared to those already existing, allowing to keep only the most restricted ones (along the lines of [4]).

Proposition 2 Let $\mathcal{C}$ be the set of constraints defining a context. Let $\mathcal{C}_1$ be the set of constraints obtained from $\mathcal{C}$ by transforming negative constraints as proposed in Algorithm 5. Let $\overline{(\mathcal{S},\tau)}$ and S_{aux} be, respectively, the database instance and the auxiliary database containing the auxiliary relations obtained by Algorithm 5 (on the basis of Definition 3.7).

For any database instance $\overline{(\mathcal{S},\tau)}$, we have

$$q(\overline{(\mathcal{S},\tau)},\mathcal{C}) = q_r(\overline{(\mathcal{S},\tau)} \cup S_{aux})$$

where $q(\overline{(\mathcal{S},\tau)},\mathcal{C})$ is the set of *valid* answers of q (with respect to $\mathcal{C}$) on $\overline{(\mathcal{S},\tau)}$ and $q_r(\overline{(\mathcal{S},\tau)} \cup S_{aux})$ is the set of answers of q_r on $\overline{(\mathcal{S},\tau)} \cup S_{aux}$ with respect to the translated set of constraints $\mathcal{C}_1$. □

PROOF:
Part 1: Let $q_r(t)$ be a result of evaluating q_r on $\overline{(\mathcal{S},\tau)} \cup S_{aux}$ with respect to $\mathcal{C}_1$. We suppose that $q_r(t)$ is not correct, *i.e.*, there is no answer t for query q on $\overline{(\mathcal{S},\tau)}$ which is valid with respect to $\mathcal{C}$.

In such a situation we know that there is a unifier θ between at least one atom in q's body and a constraint $c \in \mathcal{C}$, otherwise $q_r = q$ giving both t as an answer – which is a contradiction to our hypothesis. Thus, in this context, as θ exists, assume firstly that $c \in \mathcal{C}$ is a *positive* constraint not respected by t. The following situations are possible:

1. Let $L(u)$ be the atom in $body(q)$ for which there are homomorphism ν and σ_c (Algorithm 6, line 7) such that $\nu(body(\sigma_c(c))) = L(u)$. As t is not valid with

respect to constraint c, there exists a homomorphism h_t for which the answer of the boolean query $q() \leftarrow h_t(L(u))$ is true on instance $\overline{(\mathcal{S}, \tau)}$. Moreover, $h_t(L(u))$ triggers c (*i.e.* there is h_1 such that $h_1(body(c)) = h_t(L(u))$ and $h_1 \subseteq h'_1$) but $h'_1(head(c))$ is not true in $\overline{(\mathcal{S}, \tau)}$. However, in this case, as $h'_1(head(c))$[1] is also in the body of q_r, the answer t is not produced. A contradiction to our initial hypothesis.

2. Let $L(u)$ be the atom in $body(q)$ for which: (i) there is no homomorphism ν (Algorithm 6, line 7) such that $\nu(body(\sigma_c(c))) = L(u)$ and (ii) conditions of Lemma 3.1 hold. But in this case, as proved in the cited lemma, all answers produced by the rewritten query are valid. Thus, t is valid and we have a contradiction.

Assume now that $c \in \mathcal{C}$ is a *negative* constraint of the form $L_1(u_1) \rightarrow \bot$. In this case, c is triggered because for $L(u)$ in $body(q)$ we have $h_t(L(u)) = h_t(\sigma_c(L_1(u_1)))$ (since t is not a valid answer for q). But in this situation $\bot$ would be an atom in $body(q_r)$, and t would not be an answer for q_r, which is a contradiction to our hypothesis. The proof is similar for constraints having the form $L_1(u_1), L_2(u_2) \rightarrow \bot$ but having only constants as terms in u_1 and u_2. Lemma 3.2 completes the proof for constraints whose format is $L_1(u_1), L_2(u_2) \rightarrow \bot$ having at least a common variable in u_1 and u_2.

Part 2: We suppose that there is an answer t for query q on $\overline{(\mathcal{S}, \tau)}$ which is valid with respect to $\mathcal{C}$ and that t is not an answer for q_r.

Suppose that t is a valid answer for q with respect to $\mathcal{C}$ because no constraint is triggered (unifier θ in Algorithm 6 does not exist). In this case, we have a contradiction because $q_r = q$ and, thus, answer t is produced by both queries.

Next, as another possible situation, we assume that unifier θ (Algorithm 6) exist. Two cases have to be taken into account.

1. A positive constraint c is triggered. Let h_t is the homomorphism responsible for the production of t as an answer for query q on $\overline{(\mathcal{S}, \tau)}$. As t is valid, we know that c is triggered by an instantiation h_t of an atom $L(u)$ in $body(q)$ (*i.e.*, there is h_1 such that $h_1(body(c)) = h_t(L_u)$), and that there is a non-empty answer for query $q_{aux}() \leftarrow h'_1(head(c))$ on $\overline{(\mathcal{S}, \tau)}$ (where $h'_1 \subseteq h_1$).

   (a) Suppose $body(c)$ has no constants. In this case there is a homomorphism ν from $body(\sigma_c(c))$ to $L(u)$. From Algorithm 6, $\nu(head(\sigma_c(c))$ (or an isomorphic atom) is in the body of the rewritten version of q. As q_{aux} has an non-empty answer, we know that an instantiation for $\nu(head(\sigma_c(c))$ exists in $\overline{(\mathcal{S}, \tau)}$ allowing to obtain t as an answer for the rewritten query, a contradiction to our hypothesis.

   (b) Suppose $body(c)$ has constants. In this case, Lemma 3.1 applies.

[1] which equals $h_t(\nu(body(\sigma_c(c))))$.

2. A negative constraint is triggered. If the triggered constraint has the form $L_1(u_1) \to \bot$, we know that the evaluation of the auxiliary boolean query – whose body is $L_1(u_1)$ – is *false*. Thus, no new atoms are added to the body of q_r and the production of t is ensured. A similar argument can be used when c has the form $L_1(u_1), L_2(u_2) \to \bot$ with only constants in u_1 and u_2. Lemma 3.2 completes the proof for constraints whose format is $L_1(u_1), L_2(u_2) \to \bot$ having at least a common variable in u_1 and u_2. □

3.6.6 The whole validation process

Recall that the user's context may consist of three kinds of constraints: C_P, C_N and C_K. While the query rewriting process (Algorithm 6 and Algorithm 7) only deals with C_P and C_N (as well as the confidence degree τ_{in} of the query). Thus, the results obtained from evaluations of rewritten queries need to be verified with respect to constraints in C_K. These verification is done by using the Naive approach (cf. Algorithm 4). This overall process is depicted in Algorithm 8.

Algorithm 8: Valid candidate answers

Input : • A conjunctive query $q : \tau_{in}$ and a set of constraints $\mathcal{C} = C_P \cup C_N \cup C_K$.
• An access to the database instance $\overline{(\mathcal{S}, \tau)}$

Output: Answers of $q : \tau_{in}$ respecting $\mathcal{C}$.

```
1  Function valCandAns(q : τ_in, C):
2  AnsSet = ∅;
3  C_rw = C_P ∪ RewritingNegConstraints(C_N);
4  Q = RewriteQuery(q : τ_in, C_rw);
5  Solutions = Eval(Q, (S, τ));
6  Cache = CreateCache();
7  foreach sol ∈ Solutions where sol = (t, h_t) do
8      if Valid(sol, C_K, Cache, τ_in) then
9          AnsSet := AnsSet ∪ {t};
10 return AnsSet;
```

On line 3 of the Algorithm 8, C_{rw} is the union of C_P and the result of transforming C_N (done by the Algorithm 5). Function *RewriteQuery* on line 4 uses constraints in C_{rw} as rewriting rules to reformulate $q : \tau_{in}$ into a set Q of rewritten queries. Function *Eval* (line 5) then computes the union of the candidate answers sets for each rewritten query $q' : \tau_{in} \in Q$. These candidate answers are stored in the set Solutions. Note that each candidate answer is computed on $\overline{(\mathcal{S}, \tau)}$ with respect to C_P, C_N, τ_{in}, and they are stored in Solutions in the form of a pair (t, h_t), where h_t is the homomorphism used to produce tuple t as an answer to a query $q \in Q$. We need h_t for creating the auxiliary query that is used to validate t in the next step. Function $Valid$ verifies whether a candidate answer *sol* is valid

with respect to $\mathcal{C}_K$ and τ_{in} on $\overline{(\mathcal{S}, \tau)}$ by generating corresponding auxiliary queries. Indeed, $Valid$ implements the verification detailed in Algorithm 4 concerning $\mathcal{C}_K$ (line 18-21). A cache can be used to store results of auxiliary queries to avoid overcharging data sources (line 6,8).

Chapter 4
Experiments

In this chapter, we present the results of the experiments we carried out on the context-driven querying system presented in the previous chapter.

To the best of my knowledge, there is no similar system that supports semantic web data querying and that verifies results to obtain context-driven answers as ours. As mentioned in Chapter 2, there exist some systems that have been developed with similar technique , but either they serve for other purposes ([51, 33, 35]), or they lack the use of quality constraints to filter answers ([20, 22, 21, 55]), which is the main contribution in our work. Having different purposes, it is not suitable to compare our system with those proposals, both in terms of performance and in terms of the nature of rewritten queries. Instead, we decided to develop the two versions of our proposal, *Naive* and *Rewriting*, corresponding to the two approaches mentioned in Chapter 3, to evaluate and analyze the performance of our solutions.

4.1 General Architecture

As described in Figure 3.1 at Section 3.1, our query processing system consists of two main parts: Data validation and Data providers. All main tasks of Data validation such as performing preprocessing steps, managing queries and constraints, rewriting queries, verifying returned results are done by *Validator*. The role of Data provider is to connect to data sources and to compute answers to the queries issued from the Validator. In fact, in our system, Data provider is composed of two smaller parts:

- The first is an Integration system whose core is an ODBA system responsible for ontological query answering (knowledge-base querying).

- The second part is data sources and corresponding adapters. It consists of (i) an internal source which is managed and used by the system to store internal data and necessary intermediate results, and (ii) extended sources where queried data is stored.

The architecture of our query processing system is depicted in Figure 4.1.
Among the components of the system, Validator is the main component that contains most of the implementation of algorithms of our contribution. Therefore, experiments focus on it.

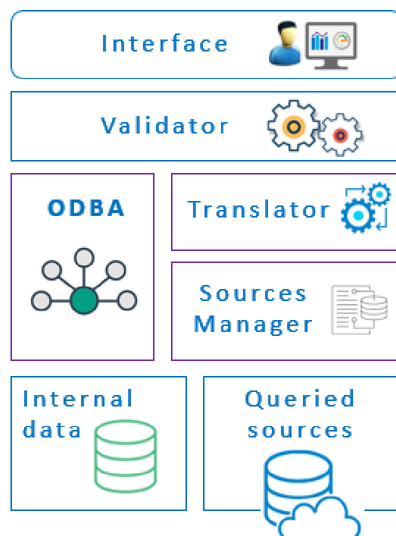

Figure 4.1 – Architecture of system

4.2 The implementation of a prototype on Graal

The system has been developed to provide an efficient solution for query processing on Semantic Web Data and to ensure the quality of the results with respect to given criteria. A prototype has been implemented in Java and developed on the basis of Graal's framework [1].

Graal is a Java toolkit dedicated to knowledge-base querying within the framework of existential rules (*e.g.* Datalog+-). The principle feature of Graal is to compute the answers to a conjunctive query over a knowledge base, a.k.a. *ontology-based query answering* . To this end, the team of Graal developed modules and libraries that allow organizing dataset with a set of rules and implementing querying algorithms on them following both well-known fashions: forward-chaining and backward-chaining (a.k.a. query rewriting). In addition, Graal has modules enabling parsing (*e.g.* parsing query, parsing rule, etc.), I/O, analyzing rule set, managing and accessing storage system, etc. Figure 4.2 [2] illustrates the main components in Graal system.

A knowledge-base in Graal is composed of a set of facts and an ontology expressed by rules. Graal supports the existential rule framework, which is known as Datalog+/-, an extension to Datalog. To be more specific, a Graal rule can be expressed by an existential rule, a rule with equality atoms in the head, or a negative rule. Interestingly, those are also the constraints defined in our system.

Graal framework defines and provides basic objects such as fact, term, variable, constant, atom, constraint, query, ontology, knowledge, etc. Those are also the main objects manipulated in our system. Moreover, fundamental algorithms such

[1] http://graphik-team.github.io/graal/
[2] http://graphik-team.github.io/graal/doc/index

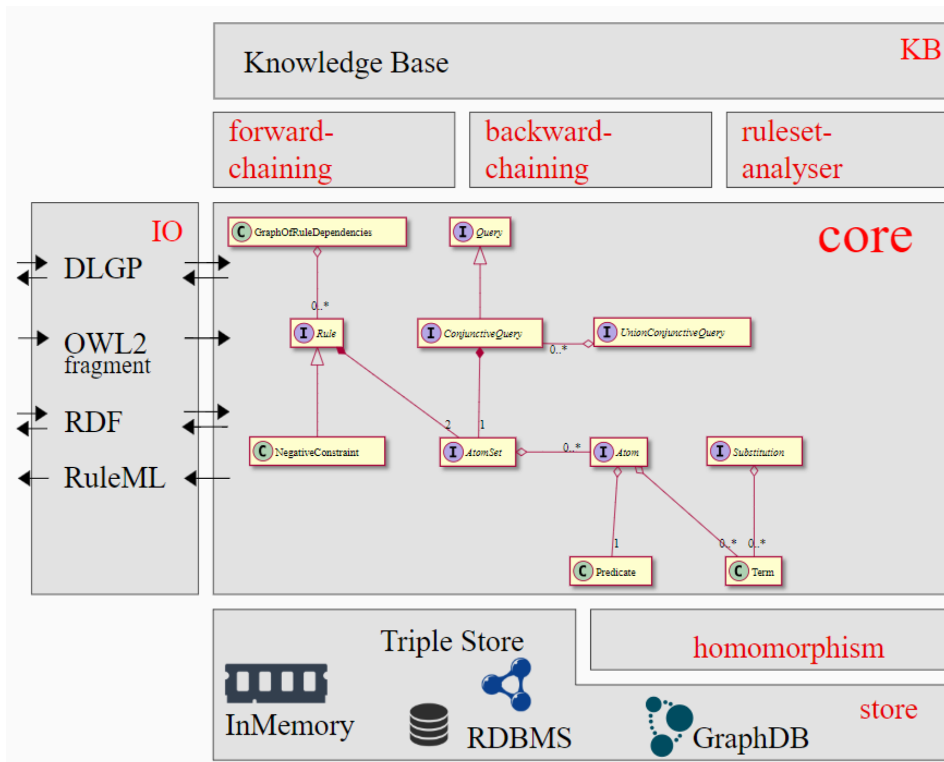

Figure 4.2 – Graal system

as substitution and homomorphism have also been studied and implemented in Graal, which helps us save a lot of effort in development.

As the core of Graal, whose principal feature is to compute the answers to a conjunctive query over a knowledge base, plays the role of the OBDA of our system, we use it as the core module in the *Data provider* part, and we add some extended modules for data source management and query translation. *Data source management* module manages information on sources as well as their confidence degrees that are set in the user-defined context. The *query translation* module is responsible for translating queries to appropriate syntax corresponding to the used data sources (*e.g.* SQL, SPARQL, etc.). In addition, we add the necessary functionality for features that are not supported by Graal, but are necessary for our validation process. All these components form the infrastructure of our system. On the top of this framework, we develop the *Validator* by implementing algorithms: Naive and Rewriting.

To this end, remember that in the procedure of querying and validating answers, there is a preprocessing step by which negative constraints are reformulated. This requires creating an auxiliary dataset, which is stored and managed as an *internal data source* of our system. We use the term *internal* for such auxiliary set in order to distinguish it from data sources in which queried data is located. For data sources, the system offers a flexible capability allowing integrate many kinds of data management systems such as the internal storage of Graal, relational database

systems, triple stores, or specialized systems such as FedX, MapReduce, etc. Each system has a corresponding adapter to serve communication between it and our system.

4.3 Different Scenarios of Experiments

To evaluate and analyze our system, we propose two scenarios of experiment corresponding to purposes as following:

- evaluate the performance of two algorithms and some factors affecting their effectiveness.
- evaluate the practicality of the solution when dealing with large unequal-trust datasets on a distributed environment.

In what follows we detail mainly the first scenario by presenting the results of experiments performed with a prototype proposing an in-memory implementation of our approach over datasets managed by Graal. We also recall the results obtained on a previous version of our validation algorithm (presented in [9]). In that version, the rewriting step takes into account only positive constraints without constants, whereas the negative constraints are treated as key constraints by sub-queries.

4.3.1 Comparing the naive and the rewriting approaches

In this experiment, we want to investigate the performance of our proposed algorithms, namely Naive and Rewriting, and to compare their efficiency. An important goal of these experiments is to analyze features that affect the computation efficiency, such as the size of datasets, the size of queries, the number and type of constraints, etc.

4.3.1.1 Experimental setting

For the purposes of the experiment, we use the LUBM benchmark [3], which describes the organizational structure of universities with 43 classes and 32 properties, and provides a generator of synthetic data with varying size.

Inspired by the 14 test queries of LUBM, we devised 7 queries and 13 (5 positive, 5 negative, and 3 key) constraints over the LUBM ontology as in Table 4.1. They are written in *Dlgp* syntax (details in [41]), an extended Datalog syntax for existential rules and Datalog+- in Graal. Queries and constraints are proposed so that they can meet the requirements and purposes of experiments. For instance, the queries spread from simple queries with few atoms ($Q1$, $Q2$) to more complex ones ($Q6$, $Q7$). There are queries whose evaluations on LUBM result in a large result sets ($Q2$, $Q4$, $Q6$), while others result in small ones ($Q5$, $Q7$). Moreover, the activation of some

[3]Lehigh University: http://swat.cse.lehigh.edu/projects/lubm/

$Q1$	$q(X,Y) \leftarrow teacherOf(X,Y).$
$Q2$	$q(X) \leftarrow Student(X).$
$Q3$	$q(X) \leftarrow GraduateStudent(X), takesCourse(X,Y).$
$Q4$	$q(X) \leftarrow Publication(X), publicationAuthor(X,Y).$
$Q5$	$q(X,Z) \leftarrow Student(X), takesCourse(X,Z),$ $teacherOf(http://www.Department0.University0.edu/AssociateProfessor0, Z).$
$Q6$	$q(X,Y) \leftarrow Student(X), takesCourse(X,Z), teacherOf(Y,Z), AssociateProfessor(Y).$
$Q7$	$q(X,Y) \leftarrow GraduateStudent(X), advisor(X,Y), takesCourse(X,Z), teacherOf(Y,Z).$
$CP1$	$teacherOf(Xprof, Xcourse) \rightarrow AssociateProfessor(Xprof).$
$CP2$	$AssociateProfessor(Xprof) \rightarrow advisor(Xstud, Xprof).$
$CP3$	$teacherOf(Xprof, http://www.Department0.University0.edu/GraduateCourse0)$ $\rightarrow FullProfessor(Xprof).$
$CP4$	$teacherOf(Xprof, http://www.Department1.University0.edu/GraduateCourse0)$ $\rightarrow AssistantProfessor(Xprof).$
$CP5$	$takesCourse(Xstud, http://www.Department0.University0.edu/GraduateCourse0)$ $\rightarrow takesCourse(Xstud, http://www.Department0.University0.edu/GraduateCourse1).$
$CN1$	$AssociateProfessor(Xprof), AssistantProfessor(Xprof) \rightarrow \bot.$
$CN2$	$AssociateProfessor(Xprof), FullProfessor(Xprof) \rightarrow \bot.$
$CN3$	$AssistantProfessor(Xprof), FullProfessor(Xprof) \rightarrow \bot.$
$CN4$	$takesCourse(Xperson, Xcourse), teacherOf(Xperson, Xcourse) \rightarrow \bot.$
$CN5$	$Student(Xperson), Professor(Xperson) \rightarrow \bot.$
$CK1$	$headOf(Xprof, Xorg1), worksFor(Xprof, Xorg2) \rightarrow Xorg1 = Xorg2.$
$CK2$	$advisor(Xstud, Xprof1), advisor(Xstud, Xprof2) \rightarrow Xprof1 = Xprof2.$
$CK3$	$doctoralDegreeFrom(Xper, Xuniv1), mastersDegreeFrom(Xper, Xuniv2)$ $\rightarrow Xuniv1 = Xuniv2.$

Table 4.1 – List of queries and constraints

constraints can trigger other constraints ($CP1$ with $CP2$, $CN1$, $CN2$; $CP2$ with $CK2$; etc.). Some queries or constraints also involve constants ($Q5$, $CP3$, $CP4$, $CP5$). Recall that, compared to other proposals using query rewriting technique in literature, our solution has handled better constants, especially in cases where constraints contain constants which do not emerge in queries. This has practical significance, because constants are commonly used in real life scenarios.

For experimental data, by means of the UBA Data Generator provided by the LUBM, we produced two ABoxes of different sizes:

- *Dataset 1* contains data of 1 university with 86,165 triples.
- *Dataset 5* has data regarding 5 universities with 515,064 triples.

Note that as produced by LUBM's generator, those datasets are consistent with the TBox of LUBM, but may be inconsistent with 13 constraints of this experiment.

As the main goal of this test is to measure the performance of the algorithms and to compare the efficiency of query rewriting with respect to separate verification queries, datasets are loaded and managed directly in the internal database system of Graal.

The experiments are carried out on an HP ZBook with a quad-core Intel i7-4800MQ processors at 2.7GHz with 32KBx4 L1 Cache, 256KBx4 L2 Cache, 6MB L3 Cache, 16GB 799MHz RAM, and a 120 GB hard drive. A 64-bit Windows 10 operating system and the 64-bit Java VM 1.8.031 constitute the software environment.

4.3.1.2 The naive validation

			Dataset 1						
	N° atoms in original query	**N° involved constr.**	**N° ans. of original query**	**N° valid ans.**	**N° eval. queries**	**Eval. time**	**Verif. time**	**Total time**
Q1	1	8	1544	523	6449	0.368	4.306	4.674
Q2	1	1	7861	7861	7862	0.109	47.769	47.878
Q3	2	2	1822	1821	3600	0.039	1.911	1.95
Q4	2	0	5939	5939	1	0.033	0.364	0.397
Q5	3	10	50	50	601	0.278	0.609	0.887
Q6	4	10	6564	6564	137977	8.218	127.19	135.408
Q7	4	9	94	21	502	14.512	0.296	14.808

(time in seconds)

Table 4.2 – Evaluation and Verification in the Naive Approach

Recall that the validation process in the Naive approach consists of two basic steps: (i) evaluate the query to obtain a candidate results set, and (ii) verify each candidate result with respect to each constraint by generating and evaluating

	N° atoms in original query	N° involved constr.	Dataset 5 N° ans.s of original query	N° valid ans.	N° eval. queries	Eval. time	Verif. time	Total time
Q1	1	8	10095	3319	42583	2.971	77.479	80.45
Q2	1	1	36682	36682	36683	0.454	659.937	660.391
Q3	2	2	11900	11900	23750	5.96	12.177	18.137
Q4	2	0	37854	37854	1	0.55	2.008	5.021
Q5	3	10	59	59	886	1.975	2.73	4.705
Q6	4	10	36008	N/A	N/A	12.617	oom.err	N/A
Q7	4	9	644	219	4328	326.689	5.244	331.933

(time in seconds)

Table 4.3 – Evaluation and Verification in the Naive Approach

sub-queries. The results of naive approach on two datasets are reported in Table 4.2 and Table 4.3. The first two columns contain the number of atoms in each tested query that indicates their complexity, and the number of constraints triggered by each one. The remaining columns in each table correspond to the experimental results of Naive algorithm in Section 3.5 on each data set. This includes information about the execution time of the tested query (column 6), the number of returned results of this evaluation (column 3), as well as the number of final valid answers of the query with respect to the constraints (column 4). This also shows the number of sub-queries needed to perform the constraint verification of returned results (column 5) and the total time required for this process (column 7).

Overall, Naive algorithm can handle well small data sets and results in all test cases. However, when performing with large datasets, some cases cannot produce results due to memory overflow. Specifically, this is the case of $Q6$. The great number of auxiliary query answers verification is the reason of this overflow. We recall that in Naive algorithm the verification of each answer with respect to a constraint can provoke not only one but a series of sub-queries due to recursive verifications of "required conditional facts" (similar to the chase process). Consequently, the number of sub-queries may be very large. Indeed, $Q6$ is the query having the longest processing time in both cases. It has 4 atoms in the body that trigger 10 constraints, and even when working with the smallest dataset it takes more than 127 seconds to execute up to 137,977 sub-queries for the validation process. Of course, as the current version stores every intermediate computed results in memory, an auxiliary database can be a better alternative solution. In spite of that, the limitation on processing large datasets is one of the main drawbacks of the naive approach.

Moreover, the experimental results show that in most of the cases the time for the verification step is the major part of the total time. This is the time used to generate and execute sub-queries. Intuitively, it depends on the size of the original

query answer set and on the number of involved constraints of the query. The more returned results and involved constraints there are, the more time it takes. Although most of sub-queries are simple queries or boolean queries that need very little executing time, they will take a lot of time and resources when large result sets are involved. Note however that, for *Q*7, the evaluating time for complex original query is significantly greater than the validating time of the small number of results (94 and 644 answers in *Dataset* 1 and *Dataset* 5 respectively).

4.3.1.3 The rewriting validation

	N° atoms in original query	**N° involved constraints**	**N° rewritten queries**	**Max n° atoms in rewritten queries**	**Rewriting time**
Q1	1	8	1	8	0.017
Q2	1	1	1	2	0.001
Q3	2	2	2	5	0.001
Q4	2	0	1	2	0
Q5	3	11	1	12	0.01
Q6	4	14	1	12	0.005
Q7	4	11	1	12	0.008

(time in seconds)

Table 4.4 – Queries and Rewritten queries (Rewriting approach)

	Dataset 1				
	N° valid answers	**Rewriting time**	**Evaluating time**	**Validating time (for CKs)**	**Total time**
Q1	523	0.017	0.473	2.094	2.584
Q2	7861	0.001	0.14	0.07	0.211
Q3	1821	0.001	0.109	0.078	0.188
Q4	5939	0	0.031	0.061	0.092
Q5	50	0.01	6.83	0.153	6.993
Q6	6564	0.005	25.296	19.351	44.652
Q7	21	0.008	1.427	0.313	0.992

(time in seconds)

Table 4.5 – Rewriting, Evaluation and Verification of Rewriting approach (Dataset 1)

We now turn our attention to the results of Rewriting approach. Recall that the processing of this approach consists of four steps:

- The *preprocessing step* that transforms negative constraints into positive constraints and computes corresponding auxiliary relations. Note that this step

	Dataset 5				
	N° valid answers	Rewriting time	Evaluating time	Validating time (for CKs)	Total time
Q1	3319	0.017	1.791	10.111	11.919
Q2	36682	0.001	1.221	0.407	1.629
Q3	11900	0.001	1.221	0.297	1.519
Q4	37854	0	0.544	0.447	1.427
Q5	59	0.01	9.37	0.249	9.629
Q6	35922	0.005	85.293	113.451	198.749
Q7	219	0.008	14.561	1.417	15.986

(time in seconds)

Table 4.6 – Rewriting, Evaluation and Verification of Rewriting approach (Dataset 5)

is performed once for each user-setting-context on a database instance. More precisely, one needs do it only when changes on negative constraints or on data sources are performed. Auxiliary relations either are materialized and managed in local system or stored as views computed at evaluating-time. To simplify, the former was realized in the current implementation. Naturally, the preprocessing time is directly proportional to the size of the dataset. The number of constraints, in general, increases after this step. This can be noticed by comparing the numbers of involved constraints in Table 4.4 with those in Table 4.3

- The *rewriting step* uses constraints (positive constraints and transformed negative constraints) as rewriting rules to reformulate the initial query into a set of rewritten queries. Notice that this step is completely independent of data in sources. The results of this step are reported in Table 4.4, which contains the following information: (i) the time need for rewriting, (ii) the number of involved constraints of the query (note that they may be different from those of the naive case), (iii) the number of the rewritten queries in the result set of this step, and (iv) the largest number of atoms in a rewritten query, which demonstrates that the more constraints are used in the rewriting procedure, the more complex are the rewritten queries (number of atoms or joins). Theoretically, the number of reformulations of a query can explode exponentially in the worst case, because each constant in a constraint can lead to two new reformulations. However, thanks to the test conditions for containment and contradiction in Query Rewriting Algorithm (line 10 Algorithm 7), only useful reformulations are accepted. Indeed, our experimental results proved that the number of rewritten queries can be very small even if the number of involved constraints is not small ($Q1$, $Q5$, $Q6$, $Q7$ in Table 4.4).

- Two last steps are *the evaluation* of rewritten queries and *the verification* of

the answers in the result set of the evaluation step. Indeed, the answers only have to be verified with respect to key constraints. The results of these steps on our two datasets are shown in Table 4.5 and Table 4.6.
The first noteworthy result is that the rewriting approach has produced answers in all tested cases. Moreover, as expected as in the naive approach, the total processing time is directly proportional to the size of the dataset.

It is worth remarking that rewritings are very fast, while the evaluation time and verification time are the major part in the total time, in all cases. Clearly the rewritten-query complexity affects the evaluation time, for instance, rewritten queries of Q6 and Q7 have 12 atoms in their body and their evaluation times on 5 universities are the biggest ones.

Evaluating time	**Dataset 1**		**Dataset 5**	
	Naïve	**rewrite**	**Naïve**	**rewrite**
Q1	0.368	0.473	2.971	1.791
Q2	0.109	0.14	0.454	1.221
Q3	0.039	0.109	5.96	1.221
Q4	0.033	0.031	0.55	0.544
Q5	0.278	6.83	1.975	9.37
Q6	8.218	25.296	12.617	85.293
Q7	14.512	1.427	326.689	14.561

(time in seconds)

Table 4.7 – Comparison of evaluating time of Naive approach and Rewrite approach

Total time	**Dataset 1**		**Dataset 5**	
	Naïve	**rewrite**	**Naïve**	**rewrite**
Q1	4.674	2.584	80.45	11.919
Q2	47.878	0.211	660.391	1.629
Q3	1.95	0.188	18.137	1.519
Q4	0.397	0.092	5.021	1.427
Q5	0.887	6.993	4.705	9.629
Q6	135.408	44.652	N/A	198.749
Q7	14.808	0.992	331.933	15.986

(time in seconds)

Table 4.8 – Comparison of total time of Naive approach and Rewrite approach

Different from Naive approach, the validation step in Rewriting approach takes into account only key constraints. Clearly, the time for this step depends on the involved key constraints and the size of the result set obtained by the evaluation of rewritten queries. Consequently, such amount of time can be a significant part in the total time, as for instance, $Q1$ and $Q6$ in Table 4.5 and Table 4.6.

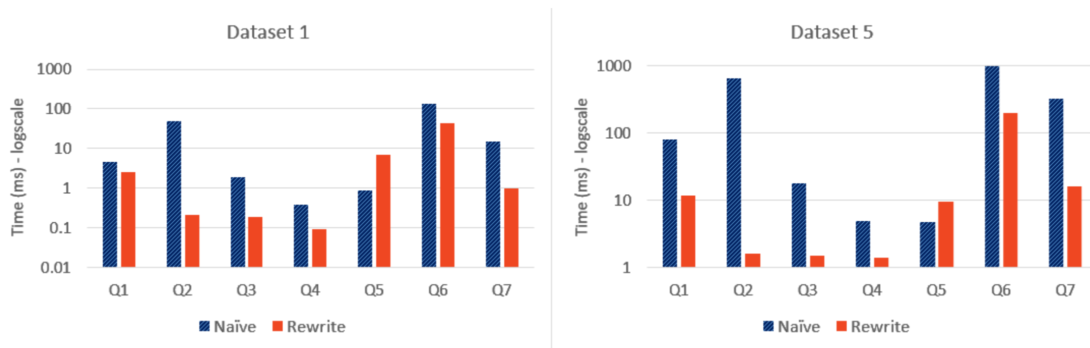

Figure 4.3 – Comparison of total time of Naive approach and Rewrite approach

4.3.1.4 Comparison

After having detailed analysis of the results of each approach, we now need to make a correlation between them. The summary of executing time of the initial queries in the naive approach and of the rewritten queries in the rewriting approach is shown in Table 4.7. In this table, it is easy to see that the executing time of rewritten queries are not always much greater than those of corresponding queries in the naive approach, even if rewritten queries are more complex after integrating the constraints. One reason is that rewritten queries can contain constants introduced by added atoms, which highly reduce the querying space. In Table 4.7, $Q7$ is the typical example for this case. Particularly, the rewritten query of $Q7$ has 12 atoms compared to 4 atoms in the original one, but its executing time is far less in both datasets.

Table 4.8 is a summary of total processing time of both approaches. It indicates one of the most meaningful observations of our experiments: *the rewriting approach is clearly far more efficient than the naive approach in most of the cases.*

Indeed, except for $Q5$, whether working on small or large dataset, whether treating simple (*e.g.* $Q1$, $Q2$) or complex (*e.g.* $Q6$, $Q7$) initial query, whether having few (*e.g.* $Q2$, $Q4$) or many (*e.g.* $Q6$, $Q7$) involved constraints, the rewriting approach always requires less processing time. $Q5$ is a special case and is unique among tested queries.In fact, the answer set of $Q5$ is small (only 50 answers on the dataset 1, or 59 on the dataset 5). As a result, although there are up to 10 involved constraints, the verification time is small (0.609 sec and 1.975 sec on dataset 1 and 5 respectively). Meanwhile, the integration of constraints has greatly increased the complexity of the rewritten query (from 3 to 12 atoms). This leads to a dramatic increase in execution time of the rewritten query compared to the original query. This amount is almost the entire total processing time, and is much greater than the verification time of this case in the naive approach. Consequently, we can conclude that: *the rewriting approach is less efficient than the naive method in some cases when dealing with a considerable number of constraints, and the evaluation of the original query gives a very small answers set in a short time.* This situation particularly deserves more investigation to establish a threshold.

4.3.2 Evaluating the use of a context-driven querying system for very large data sets

The use of our context-driven querying system over very large data sets has been the subject of our work in [9]. That work points out not only the usability of our validation on query answers but also the impact of data distribution guided by the confidence degree. Even if I, myself, have mainly worked on the validation aspects, this manuscript recalls the main results described on that research report[4].

4.3.2.1 Experimental setting

MapReduce is a simple yet powerful framework for implementing applications in large scale distributed systems without having extensive prior knowledge of issues related to data redistribution, or task allocation and fault tolerance ([48]).Most MapReduce frameworks includes *Distributed File Systems* (DFS) designed to store very large files with streaming data access patterns and data replication for fault tolerance while guaranteeing high disk I/O throughput.

We have used the open source version of MapReducem, Hadoop, developed by *The Apache Software Foundation*. Hadoop framework includes a distributed file system called HDFS[5] designed to store very large files. In MapReduce frameworks the access to data requires a full scan of input data from DFS, which may increase disk/IO and communication costs in applications involving very large datasets. Many data management frameworks have been introduced to allow efficient access to large datasets stored in a DFS. In these frameworks, only relevant columns/data can be accessed and queried using "efficient" SQL-like query languages. *Apache Hive*, *Hbase* and *Pig* are examples of such data management frameworks.

For data analysis, we have used *Hive* framework which provides data indexing and partitioning for efficient data analysis. Queries in HiveQL (a SQL-like language) are converted to a sequence of MapReduce jobs. The main motivation to use *Hive* is its ability to manage huge amount of compressed datasets. Each table can be divided into partitions (each partition corresponds to one or more HDFS buckets/splits), providing a more efficient execution of queries involving large datasets with different confidence degrees. Communication costs, HDFS disk I/O and data analysis processing time decrease because only table splits corresponding to, at least, a given confidence factor is selected for data analysis.

Figure 4.4 places Hive and MapReduce in our general architecture.

Experiments were carried out on 24 `Virtual Machines` (VMs) randomly selected from our university cluster using OpenNubula software for VMs administration. Each VM has the following characteristics: 1 Intel(R) Xeon@2.53GHz CPU, 2 Cores, 8GB of Memory and 100GB of Disk. Setting up a Hadoop cluster consists of deploying each centralised entity (namenode and jobtracker) on a dedicated

[4]Thanks to M. Bamha, specialist on MapReduce, who accepted this collaboration, results concerning data distribution were also possible.

[5]HDFS: Hadoop Distributed File System.

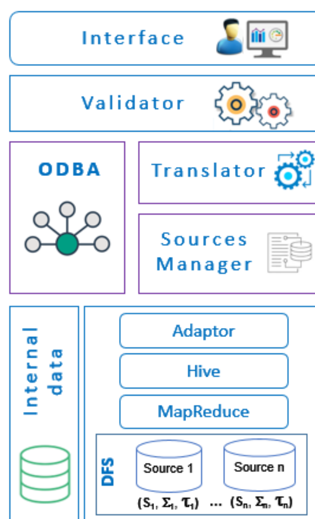

Figure 4.4 – Architecture with MapReduce

VM and co-deploying datanodes and tasktrackers on the rest of VMs. The data replication parameter was fixed to 3 in the HDFS configuration file.

4.3.2.2 The impact of data distribution using confidence factors

To evaluate the performance of our table's partitioning using confidence factors, we have compared the execution of different HiveQL queries using both partitioned and non-partitioned LUBM data sources (*w.r.t.* confidence factors). In partitioned LUBM data sources, records of each table is stored into blocks and each block contains only records corresponding to a unique confidence factor. For non-partitioned data, each block of data, of each table, may have different confidence factors.

To study the effect of the confidence factor in partitioning our LUBM benchmark (about $100GB$ of source data corresponding to approximately $8GB$ of compressed tables), we have considered confidence factors ranging from 25% to 95% (*i.e.* 0.25, 0.95).

Query	Total CPU Time	HDFS Read	Original table size
$\mathcal{Q}_1$	51.52 seconds	2709273139 Bytes (∼2.7 Gbytes)	∼19 Gbytes
$\mathcal{Q}_2$	478,13 seconds	19857918459 Bytes (∼19 Gbytes)	∼19 Gbytes

Table 4.9 – Effect of partitioned data (*w.r.t.* condidence factor) on $\mathcal{Q}$'s execution time.

Table 4.9 shows the execution of query $\mathcal{Q}$: SELECT Count(*) FROM publicationAuthor WHERE cf_factor>=85; in two different situations. We denote by $\mathcal{Q}_1$ its execution using partitioned data (*w.r.t.* confidence degree), and by $\mathcal{Q}_2$ its execution using non partitioned data. In all tests, including those presented in Table 4.9, we

notice that $\mathcal{Q}_1$ outperforms $\mathcal{Q}_2$. Execution time for $\mathcal{Q}_1$ is approximately 10 times smaller than for $\mathcal{Q}_2$. The ratio between these two execution time can be explained by the fact that in $\mathcal{Q}_1$ only relevant data (*e.g.* data corresponding a confidence factor higher than 85%) is read from HDFS whereas in $\mathcal{Q}_2$ all input data need to be read. In this scenario, HDFS disk I/O and the amount of data transmitted over the network diminish considerably, implying a reduction of the query processing time as well. However, queries with low τ_{in} are not really impacted by data partitioning according to confidence degrees because, in this case, almost all input data should be read from HDFS, anyway.

4.3.2.3 Querying under constraints on a MapReduce environment

Table 4.10 shows some experimental results obtained with a previous version of our rewriting algorithm – the one we had at the moment of that collaboration work.

	T_{1_1}	T_{1_2}	T_2	T_3	T_4
Number of CP - CN - CK	1-2-2	1-1-1	2-0-0	3-0-0	3-0-1
Query's confidence degree	85	85	60	60	60
Time to rewrite ($TRew$)	0.003	0.004	0.003	0.003	0.003
Time for first evaluation ($Time_1$)	3875	3875	3577	4533	5834
Number of answers	46	46	177371	184188	0
Number of subsidiary queries	181	120			
Time to generate subsidiary queries ($TSubQuery$)	0.397	0.334			
Number of sub queries evaluated	35	24			
Time for total sub-queries evaluation ($Time_2$)	3910	2827			
Number of validated answers	4	4			

Table 4.10 – Different steps of query validation and evaluation (time in seconds).

Results are on partitioned and compressed data (as described in Section 4.3.2.2) over 6 machines. Our performance is illustrated with a conjunctive query q having 5 joins ($6 - 8$ after rewriting). $Time_1$ expresses the maximal CPU time spent by one of our cluster machines. Lines $TRew$ and $TSubQuery$ indicate the time needed, respectively, to rewrite the given query q and to generate subsidiary queries. Both are negligible (at least when we consider the number of queries to be evaluated and the time needed for that)

The algorithm presented in [9] considers two validation steps: the rewriting step and the sub-query evaluation step. In this previous version of our approach, negative constraints were not translated into positive ones yet. For that reason, they were, together with key constraints, verified by using sub-queries. In this context, test T_1 illustrates the application of these two validation steps. The difference between T_{1_1} and T_{1_2} relies on the number of negative or key constraints. In T_{1_1}, 181 subsidiary queries are necessary for constraint validation, but only 35 are sent to the query

evaluator (the others are validated by results in the cache). All other tests apply only the rewriting step (no subsidiary query is needed). As HiveQL does not allow queries with more than one embedded sub-query, tests T_2, T_3, T_4 use at most one negative or key constraint. Tests T_2 and T_3 cannot be done by using subsidiary queries - indeed, q_1 has more than 150000 answers!

4.3.2.4 Discussion

With the work presented in [9], we have noticed that query evaluation on MapReduce evaluator faces the following troubles:

- When the number of answers for the first (rewritten) query is large, the validation via subsidiary queries is not possible. The reason is that the time required for each query evaluation on MapReduce machinery renders the whole evaluation is in general unfeasible.

- Due to HiveQL limited expression power, the rewriting option could not be adopted for the whole set of constraints. Indeed, as HiveQL does not allow queries with more than one embedded sub-query, negative constraints could not be incorporated in the query's body. This result was a motivation for us to invest in a solution for dealing with negative constraints via positive ones.

Except for this drawback, the feasibility of our approach with a MapReduce evaluator was proved.

Chapter 5

Conclusions and Perspectives

The number of currently available data (in LOD) is probably higher than ever before. Naturally, the need to use and exploit them also increase accordingly with advanced and complex tasks such as data mining, deep learning, etc. Consequently, a quality dataset is an essential precondition. As quality purposes and requirements are very diverse, and as data itself, in reality, are inconsistent and incomplete, our work plays an important role with the following **main contributions**:

The proposal of a context-driven querying system We propose a context-driven querying system which allows the satisfaction of data obtained from semantic web data providers with respect to a given *querying quality profile*. Users can establish themselves a querying quality profile by defining a set of constraints and a set of confidence degrees of data sources. The constraints comprising *positive constraints*, *negative constraints* and *key constraints* are imposed on query results. Our goal is to filter valid answers at query time instead of doing the validation on each data source, the latter being usually too heavy or unfeasible in the Web of Data context.

Two context-validation algorithms In the *Naive approach*, we first evaluate the initial query. Then, for each answer, we verify whether it respects the user's constraint. The verification is done by generating sub-queries. Experiments of this first approach show that it can perform pretty well with small datasets, but it encounters many difficulties when dealing with large datasets. Difficulties are remarked due to the explosion of the number of sub-queries which only happens when the initial query has a lot of answers and involves a large number of constraints.

The second solution, the *Rewriting approach*, attempts to minimize the number of sub-queries by integrating as much as possible involved constraints into the initial query. In that way, the result set of the new rewritten queries will correspond only to the *valid answers* of the original query. In such manner, we can do verification while doing evaluation. Compared to other works in the query rewriting context, our query rewriting algorithm takes into account both positive and negative constraints, thanks to a preprocessing step transforming the latter into positive forms, which allows us to treat them in the same way. The rewriting is based on the chase procedure. It uses constraints from left to right as rewriting rules in other to add into the query the condition to be verified. In the rewriting approach, only key constraints are verified as in the naive approach.

A sound and complete context-driven-rewriting method We have proved the sound and completeness of the rewriting algorithm, with positive constraints respecting the weak-acyclic condition. Our experiments have shown its practicability.

One important novelty of our rewriting algorithm is that it allows to integrate, into query's body, constraints which are usually ignored in other solutions [33, 42, 40]. The two typical examples are cases where constraints have atoms containing constants or duplicated variables, while in the query's body the corresponding positions contain distinct variables. In other words, these are the cases where there is no homomorphism from constraints' body to query's body.

Another very important aspect of our approach is the possibility of dealing with negative and key constraints. We recall that our key constraints allow unicity verification within a unique relation (as primary keys) or between different relations as in logical formulas: $\forall X \forall Y\ A(X, Y) \wedge \neg(\exists Z\ B(X, Z) \wedge (Y \neq Z))$ and $\forall X \forall Z\ B(X, Z) \wedge \neg(\exists Y\ A(X, Y) \wedge (Y \neq Z))$. Constraints in works such as [33, 40] are less expressive.

Experiment comparison In 86% of the situations of our experimental results, the rewriting approach outperforms the naive one. Nevertheless, as mentioned in Chapter 4, there are some factors influencing the effectiveness of the rewriting approach such as the number of involved constraints, the size of the answer set and the evaluating time of the original query, etc. All these aspects explain why in 14% of our experiments, the overall performance time of the rewriting solution is greater than the one of the naive solution. By comparing this two approaches, we could stress the importance of the rewriting method.

Several points deserve further study, and thus, we briefly present here some **perspectives** that can be envisaged extending our work.

In terms of possible optimizations: some simple practical optimizations have been implemented in the current prototype (such as detecting the presence of explicit contradictions in intermediate rewritten queries, ignoring reformulations which are not more restricted than their antecedent). Such optimizations have eliminated many meaningless rewriting-branches, thereby significantly reducing the number of reformulations in the final rewritten query set (as indicated in the experimental chapter). However, there are still many areas for improvement.

- As the most expensive step in the validation is the data querying (*i.e.* the evaluation time), for the naive approach, the implementation of a cache for valid results of previous queries is a promising solution.More precisely, during the evaluation process, intermediate valid results of boolean queries are kept at the cache. Notice that the cache mechanism can be used not only for boolean queries. It can also be extended for the rewriting approach. Obviously, a cache

update policy has to be investigated to have a trade-off between efficiency and accuracy/"fresh" of data. In a context where the evolving characteristic of data is not important (for example, infrequent changes in source data), it could be interesting to incrementally build a local warehouse for valid results of frequent queries.

- Concerning the query rewriting algorithm, the current version attempts to integrate as much as possible involved constraints into the original query. As a result, the final rewritten query set may contain excessively complex ones which usually need long execution time. Worse than that, due to the limited capability of data providers, they cannot always be translated into the query language of data provider. Deriving from the caching mechanism above, an idea arises, in which we do not entirely rewrite a query with all relevant constraints if the intermediate rewriting results of this query is "close enough" to a query that has already been rewritten previously. In this case, the evaluation and validation of current query can leverage existing results. Obviously, this needs more studies to clarify the notion "close enough" above.

In terms of experiments: we have successfully tested our solutions on a local centralized setting (with Graal) and on a large distributed data environment (via Hadoop-Hive system). From the obtained results, the following experimental works can be continued in order to extend and complement our initial purposes:

- The proposed approaches need to be tested and analyzed in a real LOD environment. Actually, we conducted experiments with a Federated-query system, namely FedX. Queries are well evaluated with a single source, which is a big instance generated with the LUBM benchmark and accessed via the Virtuoso SPARQL Endpoint. In that case our result validation proposal naturally runs (similar to those in case of Graal). Nevertheless, we could not verify it with FedX accessing more than one Virtuoso SPARQL endpoint (on two different servers), due to an error that occurs at FedX level that is not fixed yet. Fortunately, Saleem et al. in [86] introduced recently a new federated engine for SPARQL endpoint federation, namely CostFed, which has been experimentally demonstrated to be better than all the existing ones. Hence, we plan to implement our system with CostFed, thereby we can evaluate and analyze our solutions on a "closer real" LOD environment (than the setting via Hadoop-Hive).

- We have to determine in which cases the naive approach is more efficient than the rewriting one. As mentioned above, this depends on several factors, thus a thorough empirical investigation will help establish threshold-based conditions for this issue.

- Enriching the feature of the system is one of our important perspectives. In [26], aggregate functions has been theoretically considered. However, it is

necessary to completely implement and test them.

Towards more expressive constraints: in the current setting, our constraints bounds to some syntactical restrictions to guarantee the termination of chase procedure and the first-order rewritability of query answering. Particularly, constraints adopt weak-acyclicity condition, which is only a common one in many studies in literature. Clearly, considering our goal and our setting, we are interested in *data-independent chase terminations* [1], which takes only the constraints into account and not the instance in databases. In [72] we can find other proposals of this type such as *Rich acyclicity* [49], *Safe dependencies* [66], *Super Weak acyclicity* [64], *Stratification* [31], *Inductively restricted dependencies* [66], etc. The termination conditions in those proposals are based on different classes of TGDs, because the non-terminating chase sequence lies in the existentially quantified variables in the head of TGDs. As a result, choosing one of them will change our positive constraints, which leads to the need for studying and testing thoroughly for each case.

[1] In literature, there exists another class of *data-dependent chase terminations*, where the chase termination depends on a fixed database.

Bibliography

[1] Sparql endpoint status. At http://sparqles.ai.wu.ac.at/.

[2] State of the lod cloud. At http://lod-cloud.net/state/.

[3] Rdf validation workshop report, 2013. https://www.w3.org/2012/12/rdf-val/report.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

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

[6] Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. In *ISWC'11*, pages 18–34, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[8] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, June 2011.

[9] Mostafa Bamha, Jacques Chabin, Mirian Halfeld-Ferrari, Beatrice Markhoff, and Thanh Binh Nguyen. Personalized Environment for Querying Semantic Knowledge Graphs: a MapReduce Solution. Technical report, LIFO- Université d'Orléans, RR-2017-06, 2017.

[10] Cosmin Basca and Abraham Bernstein. Avalanche: Putting the spirit of the web back into semantic web querying. In *Proceedings of the 2010 International Conference on Posters & Demonstrations Track - Volume 658*, ISWC-PD'10, pages 177–180, Aachen, Germany, Germany, 2010. CEUR-WS.org.

[11] Catriel Beeri and Moshe Y. Vardi. The implication problem for data dependencies. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 73–85, London, UK, UK, 1981. Springer-Verlag.

[12] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '17, pages 37–52, New York, NY, USA, 2017. ACM.

[13] Meghyn Bienvenu. Inconsistency-tolerant conjunctive query answering for simple ontologies, 2012.

[14] Barry Bishop, Atanas Kiryakov, Damyan Ognyanov, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Factforge: A fast track to the web of data. *Semant. web*, 2(2):157–166, April 2011.

[15] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. *SPARQL Web-Querying Infrastructure: Ready for Action?*, pages 277–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[16] Marco Calautti, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna. Exploiting equality generating dependencies in checking chase termination. *Proc. VLDB Endow.*, 9(5):396–407, January 2016.

[17] Andrea Calı, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Data integration under integrity constraints. *Information Systems*, 29(2):147–163, 2004.

[18] Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Int. Res.*, 48(1):115–174, October 2013.

[19] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, pages 14–30, 2009.

[20] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

[21] Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced processing for ontological queries. *Proc. VLDB Endow.*, 3(1-2):554–565, September 2010.

[22] Andrea Cali, Domenico Lembo, and Riccardo Rosati. Query rewriting and answering under constraints in data integration systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 16–21, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[23] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Riccardo Rosati, and Marco Ruzzi. *Data Integration through DL-Lite A Ontologies*, pages 26–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] 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. *Journal of Automated Reasoning*, 39(3):385–429, 2007.

[25] Jacques Chabin, Mirian Halfeld-Ferrari, Béatrice Markhoff, and Thanh Binh Nguyen. Validating data from semantic web providers. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiří Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science*, pages 682–695, Cham, 2018. Springer International Publishing.

[26] Jacques Chabin, Mirian Halfeld-Ferrari, and Thanh Binh Nguyen. Querying Semantic Graph Databases in View of Constraints and Provenance. Technical report, LIFO- Université d'Orléans, RR-2016-02, 2016.

[27] Bizer Christian, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *IJSWIS*, 5(3):1–22, 2009.

[28] Isabel F. Cruz and Huiyong Xiao. The role of ontologies in data integration. *Engineering Intelligent Systems*, 13(4):245–252, 12 2005.

[29] Isabel F. Cruz and Huiyong Xiao. Ontology driven data integration in heterogeneous networks. In *Complex Systems in Knowledge-based Environments*, pages 75–98. 2009.

[30] Claudia d'Amato, Andrea G. B. Tettamanzi, and Tran Duc Minh. Evolutionary discovery of multi-relational association rules from ontological knowledge bases. In *EKAW*, pages 113–128. Springer International Publishing, 2016.

[31] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. The chase revisited. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 149–158, 2008.

[32] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 459–470, 1999.

[33] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Rec.*, 35(1):65–73, March 2006.

[34] Alin Deutsch and Val Tannen. Reformulation of xml queries and constraints. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 225–241, London, UK, UK, 2002. Springer-Verlag.

[35] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, May 2005.

[36] Luis Galárraga, Simon Razniewski, Antoine Amarilli, and Fabian M. Suchanek. Predicting Completeness in Knowledge Bases. In *WSDM*, pages 375–383, 2017.

[37] Olaf Görlitz and Steffen Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *COLD'11*, pages 13–24, Aachen, Germany, Germany, 2010. CEUR-WS.org.

[38] Georg Gottlob, Thomas Lukasiewicz, Maria Vanina Martinez, and Gerardo I. Simari. Query answering under probabilistic uncertainty in datalog+ / - ontologies. *Ann. Math. Artif. Intell.*, 69(1):37–72, 2013.

[39] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, pages 2–13, 2011.

[40] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *CoRR*, abs/1405.2848, 2014.

[41] Montpellier GraphIK Team, LIRMM-Inria. Brief overview of the existential rule framework. At https://graphik-team.github.io/graal/papers/framework-en.pdf, 2012.

[42] Luca Grieco, Domenico Lembo, Riccardo Rosati, and Marco Ruzzi. Consistent query answering under key and exclusion dependencies: Algorithms and experiments. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 792–799, New York, NY, USA, 2005. ACM.

[43] Thomas R Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *In International Journal Human-Computer Studies*, 43:907–928, 1993.

[44] Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International journal of human-computer studies*, 43(5):907–928, 1995.

[45] Olaf Hartig. *SPARQL for a Web of Linked Data: Semantics and Computability*, pages 8–23. 2012.

[46] Olaf Hartig. SQUIN: A Traversal Based Query Execution System for the Web of Linked Data. In *SIGMOD*, pages 1081–1084, New York, NY, USA, 2013. ACM.

[47] Olaf Hartig. Querying a web of linked data: foundations and query execution. *SIGWEB Newsletter*, 2014(Autumn):3:1–3:2, 2014.

[48] M. Al Hajj Hassan, M. Bamha, and Frédéric Loulergue. Handling data-skew effects in join operations using mapreduce. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 145–158, 2014.

[49] Andre Hernich and Nicole Schweikardt. Cwa-solutions for data exchange settings with target dependencies. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, pages 113–122, New York, NY, USA, 2007. ACM.

[50] Ian Horrocks, Patrick Hayes, and Peter Patel-Schneider. OWL web ontology language semantics and abstract syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[51] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. Complete yet practical search for minimal query reformulations under constraints. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1015–1026, New York, NY, USA, 2014. ACM.

[52] Holger Knublauch. Spin - modeling vocabulary, February 2011. http://www.w3.org/Submission/2011/SUBM-spin-modeling-20110222/.

[53] George Konstantinidis and José Luis Ambite. Optimizing the chase: Scalable data integration under constraints. *Proc. VLDB Endow.*, 7(14):1869–1880, October 2014.

[54] Dimitris Kontokostas and Holger Knublauch. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[55] Domenico Lembo, Maurizio Lenzerini, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. Inconsistency-tolerant query answering in ontology-based data access. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:3 – 29, 2015.

[56] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[57] Thomas Lukasiewicz, Maria Vanina Martinez, and Gerardo I. Simari. Inconsistency handling in datalog+/- ontologies. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pages 558–563, 2012.

[58] Thomas Lukasiewicz, Maria Vanina Martinez, and Gerardo I. Simari. Inconsistency-tolerant query rewriting for linear datalog+/–. In Pablo Barceló and Reinhard Pichler, editors, *Datalog in Academia and Industry*, pages 123–134, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[59] Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura. *ADERIS: An Adaptive Query Processor for Joining Federated SPARQL Endpoints*, pages 808–817. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[60] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Trans. Database Syst.*, 4(4):455–469, December 1979.

[61] David Maier, Yehoshua Sagiv, and Mihalis Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, October 1981.

[62] Nicolas Marie and Fabien L. Gandon. Survey of Linked Data Based Exploration Systems. In *IESD Workshop at ISWC*, 2014.

[63] Béatrice Markhoff, Thanh Binh Nguyen, and Cheikh Niang. When it comes to querying semantic cultural heritage data. In *New Trends in Databases and Information Systems - ADBIS 2017 Short Papers and Workshops, AMSD, BigNovelTI, DAS, SW4CH, DC, Nicosia, Cyprus, September 24-27, 2017, Proceedings*, pages 384–394, 2017.

[64] Bruno Marnette. Generalized schema-mappings: From termination to tractability. In *Proceedings of the Twenty-eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 13–22, New York, NY, USA, 2009. ACM.

[65] Giansalvatore Mecca, Paolo Papotti, and Salvatore Raunich. Core schema mappings. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 655–668, New York, NY, USA, 2009. ACM.

[66] Michael Meier, Michael Schmidt, and Georg Lausen. On chase termination beyond stratification. *Proc. VLDB Endow.*, 2(1):970–981, August 2009.

[67] Gabriela Montoya, Luis-Daniel Ibáñez, Hala Skaf-Molli, Pascal Molli, and Maria-Esther Vidal. *SemLAV: Local-As-View Mediation for SPARQL Queries*, pages 33–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[68] Gabriela Montoya, Maria-Esther Vidal, and Maribel Acosta. A heuristic-based approach for planning federated sparql queries. In *Proceedings of the Third International Conference on Consuming Linked Data - Volume 905*, COLD'12, pages 63–74, Aachen, Germany, Germany, 2012. CEUR-WS.org.

[69] Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between OWL and relational databases. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(2), 2009.

[70] Andriy Nikolov, Andreas Schwarte, and Christian Hütter. *FedSearch: Efficiently Combining Structured Queries and Full-Text Search in a SPARQL Federation*, pages 427–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[71] Andreas Nolle and German Nemirovski. ELITE: an entailment-based federated query engine for complete and transparent semantic data integration. In *Informal Proceedings of the 26th International Workshop on Description Logics, Ulm, Germany, July 23 - 26, 2013*, pages 854–867, 2013.

[72] Adrian Onet. The chase procedure and its applications in data exchange. In *Data Exchange, Integration, and Streams*, pages 1–37. 2013.

[73] M. Tamer Özsu. A survey of rdf data management systems. *Front. Comput. Sci.*, 10(3):418–432, June 2016.

[74] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[75] Rachel Pottinger and Alon Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3):182–198, 2001.

[76] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. Shape expressions: An rdf validation and transformation language. In *Proceedings of the 10th International Conference on Semantic Systems*, SEM '14, pages 32–40, New York, NY, USA, 2014. ACM.

[77] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. In *ESWC*, pages 524–538, Berlin, Heidelberg, 2008. Springer-Verlag.

[78] Nur Aini Rakhmawati, Jürgen Umbrich, Marcel Karnstedt, Ali Hasnain, and Michael Hausenblas. Querying over federated SPARQL endpoints - A state of the art survey. *CoRR*, abs/1306.1723, 2013.

[79] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Logical query optimization by proof-tree transformation. *Journal of Computer and System Sciences*, 47(1):222 – 248, 1993.

[80] Riccardo Rosati. On the complexity of dealing with inconsistency in description logic ontologies. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1057–1062. AAAI Press, 2011.

[81] Riccardo Rosati, Marco Ruzzi, Mirko Graziosi, and Giulia Masotti. Evaluation of techniques for inconsistency handling in owl 2 ql ontologies. In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*, ISWC'12, pages 337–349, Berlin, Heidelberg, 2012. Springer-Verlag.

[82] Arthur G. Ryman, Arnaud Le Hors, and Steve Speicher. OSLC resource shape: A language for defining constraints on linked data. In *LDOW*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[83] Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A fine-grained evaluation of sparql endpoint federation systems. *Semantic Web*, (Preprint):1–26, 2015.

[84] Muhammad Saleem and Axel-Cyrille Ngonga Ngomo. *HiBISCuS: Hypergraph-Based Source Selection for SPARQL Endpoint Federation*, pages 176–191. 2014.

[85] Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, Helena F. Deus, and Manfred Hauswirth. *DAW: Duplicate-AWare Federated Query Processing over the Web of Data*, pages 574–590. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[86] Muhammad Saleema, Alexander Potockia, Tommaso Sorua, Olaf Hartigc, and Axel-Cyrille Ngonga Ngomob. Costfed: Cost-based query optimization for sparql endpoint federation.

[87] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, pages 601–616, 2011.

[88] Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[89] Balder ten Cate, Laura Chiticariu, Phokion Kolaitis, and Wang-Chiew Tan. Laconic schema mappings: Computing the core with sql queries. *Proc. VLDB Endow.*, 2(1):1006–1017, August 2009.

[90] Moshe Y. Vardi. Inferring multivalued dependencies from functional and join dependencies. *Acta Informatica*, 19(4):305–324, Sep 1983.

[91] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. Ontology-based integration of information-a survey of existing approaches. In *IJCAI-01 workshop: ontologies and information sharing*, volume 2001, pages 108–117. Citeseer, 2001.

[92] Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. LHD: optimising linked data query processing using parallelisation. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013.

[93] Gerhard Weikum, Johannes Hoffart, and Fabian M. Suchanek. Ten years of knowledge harvesting: Lessons and challenges. *IEEE Data Eng. Bull.*, 39(3):41–50, 2016.

[94] David Wood, Markus Lanthaler, and Richard Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

Thanh Binh NGUYEN

L'interrogation du web de données garantissant des réponses valides par rapport à des critères donnés

Résumé : Le terme Linked Open Data (LOD) (ou données ouvertes liées) a été introduit pour la première fois par Tim Berners-Lee en 2006. Depuis, les LOD ont connu une importante évolution. Aujourd'hui, nous pouvons constater les milliers de jeux de données présents sur le Web de données. De ce fait, la communauté de recherche s'est confrontée à un certain nombre de défis concernant la récupération et le traitement de données liées.
Dans cette thèse, nous nous intéressons au problème de la qualité des données extraites de diverses sources du LOD et nous proposons un système d'interrogation contextuelle qui garantit la qualité des réponses par rapport à un contexte spécifié par l'utilisateur. Nous définissons un cadre d'expression de contraintes et proposons deux approches : l'une naïve et l'autre de réécriture, permettant de filtrer dynamiquement les réponses valides obtenues à partir des sources éventuellement non-valides, ceci au moment de la requête et non pas en cerchant à les valider dans les sources des données. L'approche naïve exécute le processus de validation en générant et en évaluant des sous-requêtes pour chaque réponse candidate en fonction de chaque contrainte. Alors que l'approche de réécriture utilise les contraintes comme des règles de réécriture pour reformuler la requête en un ensemble de requêtes auxiliaires, de sorte que les réponses à ces requêtes réécrites ne sont pas seulement les réponses de la requête initiale mais aussi des réponses valides par rapport à toutes les contraintes intégrées. La preuve de la correction et de la complétude de notre système de réécriture est présentée après un travail de formalisation de la notion de réponse valide par rapport à un contexte. Ces deux approches ont été évaluées et ont montré la praticabilité de notre système.
Ceci est notre principale contribution: nous étendons l'ensemble de systèmes de réécriture déjà connus (Chase, C&BC, PerfectRef, Xrewrite, etc.) avec une nouvelle solution efficace pour ce nouveau défi qu'est le filtrage des résultats en fonction d'un contexte utilisateur. Nous généralisons également les conditions de déclenchement de contraintes par rapport aux solutions existantes, en utilisant la notion de one-way MGU.
Mots clés : Web sémantique, réécriture de requêtes, contrainte de qualité utilisateur, provenance des données

Querying the Web of Data guaranteeing valid answers with respect to given criteria

Abstract: The term Linked Open Data (LOD) is proposed the first time by Tim Berners-Lee since 2006. Since then, LOD has evolved impressively with thousands datasets on the Web of Data, which has raised a number of challenges for the research community to retrieve and to process LOD.
In this thesis, we focus on the problem of quality of retrieved data from various sources of the LOD and we propose a context-driven querying system that guarantees the quality of answers with respect to the quality context defined by users. We define a fragment of constraints and propose two approaches: the naive and the rewriting, which allows us to filter dynamically valid answers at the query time instead of validating them at the data source level. The naive approach performs the validation process by generating and evaluating sub-queries for each candidate answer w.r.t. each constraint. While the rewriting approach uses constraints as rewriting rules to reformulate query into a set of auxiliary queries such that the answers of rewritten-queries are not only the answers of the query but also valid answers w.r.t. all integrated constraints. The proof of the correction and completeness of our rewriting system is presented after formalizing the notion of a valid answers w.r.t. a context. These two approaches have been evaluated and have shown the feasibility of our system.
This is our main contribution: we extend the set of well-known query-rewriting systems (Chase, Chase & backchase, PerfectRef, Xrewrite, etc.) with a new effective solution for the new purpose of filtering query results based on constraints in user context. Moreover, we also enlarge the trigger condition of the constraint compared with other works by using the notion of one-way MGU.
Keywords: Semantic Web Data, Query rewriting, User Quality constraint, Data provenance

Centre Val de Loire UNIVERSITÉ Spirit of the Loire Valley

Laboratoire d'Informatique Fondamentale d'Orléans
Bat. 3IA, Université d'Orléans
Rue Léonard de Vinci, B.P. 6759
F-45067 ORLEANS Cedex 2

LABORATOIRE D'INFORMATIQUE LIFO FONDAMENTALE D'ORLEANS