Integrating Heterogeneous Data Sources in the Web of Data

Franck Michel

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE STIC :

Sciences et Technologies de l'Information et de la Communication

UNITÉ DE RECHERCHE :

Laboratoire I3S, UMR 7271 CNRS Université Côte d'Azur, équipe SPARKS

THÈSE DE DOCTORAT

Présentée en vue de l'obtention du grade de
Docteur en Informatique
de

UNIVERSITÉ CÔTE D'AZUR

par

Franck MICHEL

INTEGRATING HETEROGENEOUS DATA SOURCES IN THE WEB OF DATA

Dirigée par Johan MONTAGNAT, Directeur de Recherche, CNRS
et codirigée par Catherine FARON-ZUCKER, Maître de Conférences, Université Côte d'Azur

Soutenue le 3 Mars 2017
Devant le jury composé de

Rapporteurs :	Oscar CORCHO	Professeur, Facultad de Informática Boadilla del Monte
	Marie-Christine ROUSSET	Professeur, Université de Grenoble
Directeur :	Johan MONTAGNAT	Directeur de Recherche, CNRS
Codirecteur :	Catherine FARON-ZUCKER	Maître de Conférences, Université Côte d'Azur
Président :	Fabien GANDON	Directeur de Recherche, Inria
Examinateurs :	Cécile CALLOU	Maître de Conférences, Muséum National d'Histoire Naturelle
	Pascal MOLLI	Professeur, Université de Nantes
	Pascal NEVEU	Ingénieur de Recherche, Institut National de Recherche en Agronomie

Version	Date	Description
v1	2017-04-14	Initial version with changes requested by reviewers
v2	2017-10-08	Fix misc. typos and add the xrr:pushdown feature to the xR2RML language

Integrating Heterogeneous Data Sources in the Web of Data

To a great extent, RDF-based data integration as well as the Web of Data depend on the ability to reach out to legacy data locked in data silos where they are invisible to the Web. In the last 15 years, various works have tackled the problem of translating structured data into the Resource Description Framework (RDF), starting with relational databases (RDB), spreadsheets and the XML data format. Meanwhile, the overwhelming success of NoSQL databases has made the database landscape more diverse than ever. So far, though, these databases remain inaccessible to RDF-based data integration systems, and although the data they host may be of interest to a large audience, they remain invisible to the Web of Data. Hence, to harness the potential of NoSQL databases and more generally non-RDF data sources, the objective of this thesis was to enable RDF-based data integration over heterogeneous databases and, in particular, to bridge the gap between the Semantic Web and the NoSQL family of databases.

Firstly, we proposed a generic mapping language, xR2RML, able to describe the mapping of several types of databases into an arbitrary RDF representation. This language relies on and extends previous works on the translation of RDBs, CSV and XML into RDF. Secondly, we proposed to use such an xR2RML mapping either to materialize RDF data or to dynamically evaluate SPARQL queries against the native database. To spur the development of SPARQL interfaces over legacy databases, we propose a two-step approach. The first step performs the translation of a SPARQL query into a pivot abstract query based on the xR2RML mapping of the target database to RDF. In the second step, the abstract query is translated into a concrete query, taking into account the specificities of the database query language. Great care is taken of the query optimization opportunities, both at the abstract and the concrete levels. To demonstrate the effectiveness of our approach, we developed a prototype implementation for MongoDB, the popular NoSQL document store. We have validated the method using a real-life use case in Digital Humanities.

Keywords: Data Integration, legacy data, Web of Data, virtual RDF store, xR2RML, SPARQL, MongoDB

Intégrer des Sources de Données Hétérogènes dans le Web de Données

Dans une large mesure, l'intégration de données basée sur le format RDF (Resource Description Framework), ainsi que le succès du Web de Données, reposent sur notre capacité à atteindre les données stockées dans des silos d'où elles restent invisibles pour le Web. Durant les quinze dernières années, différents travaux ont entrepris d'exposer des données structurées au format RDF, notamment les données de bases relationnelles, et les formats CSV, TSV et XML. Dans le même temps, le marché des bases de données (BdD) est devenu très hétérogène avec le succès massif des BdD dites NoSQL. Jusqu'ici pourtant, celles-ci restent inaccessibles aux systèmes d'intégration de données basés sur RDF. De plus, bien que les données qu'elles hébergent puissent potentiellement intéresser un large public, celles-ci restent invisibles depuis le Web de Données. Aussi, afin d'exploiter le potentiel des BdD NoSQL et plus généralement des sources non-RDF, l'objectif de cette thèse est de permettre l'intégration de sources de données hétérogènes basée sur le format RDF, et en particulier établir des ponts entre le Web Sémantique et la famille des BdD NoSQL.

En premier, nous proposons un langage générique, xR2RML, permettant de décrire l'alignement (mapping) de sources de données de types variés vers une représentation RDF arbitraire. Ce langage étend des travaux précédents sur la traduction de sources relationnelles, CSV et XML en RDF. Puis nous proposons d'utiliser une telle description xR2RML soit pour matérialiser les données RDF, soit pour évaluer dynamiquement des requêtes SPARQL sur la base native. Afin d'encourager le développement d'interfaces SPARQL à des BdD existantes, nous proposons une approche en deux étapes. La première effectue la traduction d'une requête SPARQL en une requête pivot, abstraite, en se basant sur le mapping xR2RML de la BdD cible vers RDF. Dans la seconde étape, la requête abstraite est traduite en une requête concrète, prenant en compte les spécificités du langage de requête de la BdD. Un souci particulier est apporté aux possibilités d'optimisation des requêtes, tant au niveau abstrait que concret. Pour démontrer l'applicabilité de notre approche, nous avons développé un prototype pour une base NoSQL populaire : MongoDB, et nous avons validé la méthode dans un cas d'utilisation réel issu du domaine des humanités numériques.

Mots-clés : Intégration de données, données historiques, Web de Données, entrepôt RDF virtuel, xR2RML, SPARQL, MongoDB

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1. Introduction

1.1 Motivations

The Resource Description Framework (RDF) [Cyganiak et al., 2014] is increasingly adopted as the pivot format for integrating heterogeneous data sources. Several reasons can be pointed out to explain this trend. First, RDF offers a unified data model (directed labelled graphs). Second, it allows building upon countless domain knowledge formalizations, in the form of vocabularies, thesauri and ontologies that can be freely reused and extended. Third, RDF-based data integration systems benefit from the reasoning capabilities offered by the Semantic Web technologies[1], that are backed by extensive theoretical works. Lastly, RDF makes it possible to leverage the huge knowledge base represented by the Web of Data, thereby opening up opportunities to discover related data sets, enrich data, and create added value by mashing up all this information (see Figure 1).

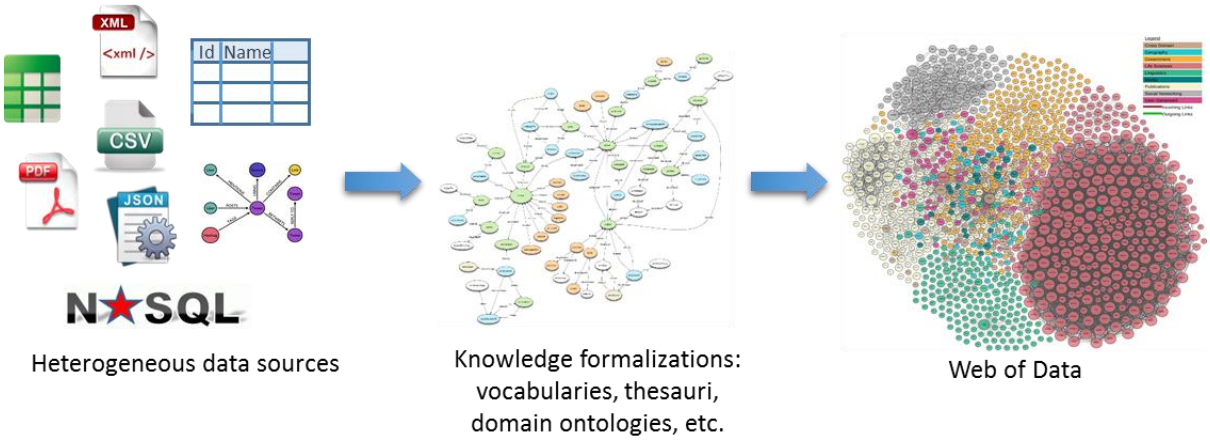


Figure 1: Integration of heterogeneous data with the Web of Data using knowledge formalizations

Nevertheless, RDF-based data integration needs are generally concerned with legacy data that are not natively stored as RDF. Typically, the Web of Data is progressively emerging as more institutions and organizations publish their data following the *Linked Open Data* principles [Berners-Lee, 2006], but huge amounts of data remain locked in silos where they are not accessible to the Web nor to data integration systems. These data, often referred to as the deep Web [He et al., 2007], typically consist of legacy relational databases and files in various data formats, usually queried through Web forms or Web services. Such data sources are hardly linked with each other and hardly indexed by search engines.

Consequently, harnessing these legacy data sources either to perform RDF-based integration or to publish Linked Data on the Web of Data requires methods to translate heterogeneous data into an RDF representation. Since the early 2000's, much work has investigated such methods. In one way or another, they all rely on the explicit or implicit description of a mapping that instructs how to translate

[1] https://www.w3.org/standards/semanticweb/

each data item from its original format into an RDF representation. In this respect, relational databases (RDB) have caught much attention due to their dominant position [Spanos et al., 2012; Sequeda et al., 2011; Michel et al., 2014]. These works lead to the publication, in 2012, of the R2RML W3C recommendation on the description of RDB-to-RDF mappings [Das et al., 2012]. Other significant works have focused on a handful of structured data formats such as XML and CSV, the latter is the object of a recent W3C recommendation [Tandy et al., 2015].

Graph Materialization vs. Query Rewriting. Two approaches generally apply when it comes to produce RDF from legacy data. The RDF **graph materialization** is the static transformation of a data source into RDF. Mapping rules are applied exhaustively to the content of the data source to create an RDF graph. For this reason, it is also referred to as "graph dump", "graph extraction" or "RDF dump". The resulting RDF graph is typically loaded into a triple store and accessed through a query engine supporting the SPARQL query language [Harris & Seaborne, 2013] or as Linked Data by dereferencing URIs [Heath & Bizer, 2011]. This whole process is called Extract-Transform-Load (ETL), as an analogy to data warehousing practices. This approach has the advantage of enabling further processing, analysis or reasoning. Indeed, since the RDF data is made available at once, third party reasoning tools can be used to apply complex entailments. Not surprisingly, the main drawback is that the materialized RDF graph may rapidly be outdated if the data source is updated frequently. A workaround is to run the extraction process periodically, but in the context of very large data sets, a compromise must be found between the cost (in time, memory and CPU) of materializing and reloading the graph, and the extent to which outdated data is acceptable. Another option is to track down the changes in the data source and update the graph accordingly, such as the DBpedia Live initiative[2].

Alternatively, the **virtual RDF graph** approach (*aka*. **query rewriting**, **dynamic access**) accesses legacy data on-the-fly using the SPARQL query language. The data remains located in the legacy data source. Based on the mapping of the data source to RDF, a SPARQL query on the virtual RDF graph is rewritten into a query that the data source can process, and evaluated at run-time. In practice, the query rewriting approach scales better to large data sets that would hardly support materialization, and it guarantees data freshness. On the other hand, query rewriting entails overheads that can penalize query performance. In particular, the implementation of entailment regimes may generate many queries and ultimately lead to very poor performances [Sahoo et al., 2009].

NoSQL Databases. During the last decade, the database landscape has become more heterogeneous than ever as the emerging NoSQL movement was gathering momentum. Initially confined to serve as the core system of Big Data applications, NoSQL databases are being increasingly adopted as general-purpose, commonplace databases. This trend is fostered by their open source licenses, their lightweight, easy-to-start packaging (for some of them), and the fact that the most popular ones are well supported in common programming languages and frameworks. Consequently, despite controversy arguments about the lack of theoretical background or the comparison with parallel RDBs [Pavlo et al., 2009; Floratou et al., 2014; Stonebraker et al., 2010; Dean & Ghemawat, 2010; Stonebraker, 2012], the success of NoSQL databases is no longer questioned today. More and more companies and institutions increasingly use them to store valuable data related to all sorts of domains.

[2] DPpedia Live : http://wiki.dbpedia.org/online-access/DBpediaLive

So far, though, these data remain inaccessible to RDF-based data integration systems, and although some of them may be of interest to a large audience, they remain invisible to the Web of Data.

1.2 Objectives

Objectives of this thesis. Thus, we think it is increasingly important to study how to harness the potential of NoSQL databases and more generally non-RDF data sources, to enable RDF-based data integration as well as to populate the Web of Data. Hence the objectives of this thesis:

> *In this thesis, we tackle the challenges of enabling RDF-based data integration over heterogeneous databases and, in particular, we propose a method to bridge the gap between the Semantic Web and the NoSQL family of databases.*

This raises multiple challenges though, since the Semantic Web technologies, that underpin the Web of Data, and the NoSQL movement, are two separate worlds based on very different paradigms. Thereby, our work consists of three main contributions.

> ***Contribution 1****. To foster the translation of heterogeneous (legacy) data sources into RDF, we propose a generalized mapping language, xR2RML, able to describe the mapping of data items from heterogeneous types of databases into an arbitrary RDF representation.*

The xR2RML language may be operationalized to produce an RDF graph representing the data source, following the graph materialization approach. However, as we mentioned above, the size of some large data sets and the need for up-to-date data may require adopting the virtual graph approach. Thus, to spur the development of SPARQL interfaces for legacy data sources, we propose a two-step approach to execute SPARQL queries over heterogeneous databases. In the second and third contributions, we define a pivot abstract query language to accommodate the multiplicity of databases, the translation from SPARQL to that abstract query language, and the translation from that abstract query language to the concrete query language of a target database.

> ***Contribution 2****. We propose the database-independent translation of a SPARQL query into a pivot abstract query based on the xR2RML mapping of the database to RDF. The contribution includes the definition of the abstract query language and the xR2RML-based method to translate a SPARQL query into an abstract query. Furthermore, great care is taken of enforcing optimizations at the abstract query level, such that only database-specific optimizations are left to the subsequent stage.*

This first query rewriting step aims to achieve as much of the translation process as possible, regardless of the target database. Query optimizations at the abstract query level alleviate the subsequent translation step, thereby reducing the complexity of database adaptors.

***Contribution 3**. In the second step, we translate an abstract query into a concrete query by taking into account the specific database query capabilities. We demonstrate the effectiveness of our method in the context of a popular NoSQL database: MongoDB. Despite the discrepancy between the expressiveness of SPARQL and the MongoDB query language, we show that it is always possible to translate an abstract query into MongoDB queries, and that the rewriting yields all the correct answers.*

The application to MongoDB underlines that, in some cases, optimizations enforced at the abstract query level could not have been enforced in the subsequent step due to limitations of the target query language.

***Contribution 4**. Finally, we propose an open source prototype implementation of the method, that we evaluated in a real-world use case.*

1.3 Thesis Outline and Publications

This manuscript consists of the following chapters:

Chapter 2: As an addition to the introduction, we provide an in-depth description of the context of this thesis: this includes a description of the concept and evolution of the open data movement, the Linked Data and Web of Data and their relationship with open data, and the history and fundamentals of the NoSQL family of databases.

Chapter 3: In this state of the art, we remind principles of data integration and their application to the contexts of ontology-based query answering and ontology-based data access. Then, we underline the key role of mappings, and we review previous works on the definition of mappings from varying types of data sources to RDF.

Chapter 4: In this chapter, we investigate what properties a generalized mapping language should have to enable the mapping of an extensible scope of databases to RDF, in a flexible and transparent manner. We formalize these properties as a set of six requirements and we evaluate some of the mapping languages reviewed in Chapter 3 against those requirements.

Chapter 5: Following the analysis conducted in Chapter 4, our first contribution consists of the specification of xR2RML, a generalized mapping language that builds upon and extends previous works.

This work was published in the proceedings of the WebIST 2015 international conference [Michel et al., 2015]. It was one of the best-paper nominees, and an extended version was published in Springer's Lecture Notes in Business Information Processing [Michel et al., 2016a].

Chapter 6: To foster the development of SPARQL interfaces to heterogeneous databases, our second contribution is the first step of a two-step approach to execute SPARQL queries over heterogeneous databases. Chapter 6 defines an abstract query language meant to abstract the approach from the details of specific databases, and devises a method to

translate a SPARQL query into a pivot abstract query based on the xR2RML mapping of the database to RDF.

The work presented in this chapter was published in the proceedings of the WebIST 2016 international conference [Michel et al., 2016b].

Chapter 7: The second step of our approach enacts the part of the translation that specifically depends on the target database. To illustrate the effort it takes to translate from SPARQL towards a query language that is far less expressive than SPARQL, we implement the method in the context of a popular NoSQL database: MongoDB. In this third contribution, we devise a translation from abstract queries into MongoDB queries.

The work presented in this chapter was published in the proceedings of the DEXA 2016 international conference [Michel et al., 2016c].

Chapter 8: To illustrate the effectiveness of the whole approach, we carried out an experimental evaluation in the context of a real-world use case: the goal is to translate a taxonomical reference (TAXREF) into a SKOS thesaurus. In this chapter, we elaborate on how we use xR2RML to map the MongoDB database hosting TAXREF into the chosen SKOS representation. We briefly present the software prototype that we implemented, and we present the results of performance measures with respect to the RDF graph materialization on the one hand, and the SPARQL-to-MongoDB query rewriting on the other hand.

The modelling of TAXREF as a SKOS thesaurus and its operationalization with xR2RML was published in a workshop of the ESWC 2015 conference [Callou et al., 2015].

Chapter 9: The last chapter of this manuscript provides conclusive remarks and suggests leads for future works.

1.4 Conventions

Throughout this manuscript, examples containing snippets of RDF and SPARQL assume the following namespace prefix bindings:

Prefix	IRI
rr	http://www.w3.org/ns/r2rml#
rml	http://semweb.mmlab.be/ns/rml#
ql	http://semweb.mmlab.be/ns/ql#
xrr	http://www.i3s.unice.fr /ns/xr2rml#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs	http://www.w3.org/2000/01/rdf-schema#
xsd	http://www.w3.org/2001/XMLSchema#
ex	http://example.com/ns#

Chapter 2. In-Depth Context

This chapter is intended to provide the interested reader with a more in-depth understanding of the context of this thesis.

From a general standpoint, our work is concerned with RDF-based data integration. Producing and consuming linked data is a prominent application thereof. Although not all linked data is meant to be open, open data is admittedly a major source of linked data insofar as it contributes to the building of worldwide knowledge commons. Thus, below we first describe the philosophy and evolution of the open data movement and its connection with the concept of commons. Then, we remind the principles of Linked Data and the Web of Data, and their relationship with open data.

In the last section of this chapter, we remind the fundamentals of the NoSQL family of databases and the reasons for its emergence.

2.1 From the Commons to the Open Data

During the Middle Age, the United Kingdom initiated a movement called the Enclosure Acts[3] that created legal property rights to land that was previously considered as a common good. Slowly, Europe transitioned from a traditional subsistence rural economy to a modern agricultural economy, driven by nascent markets where exclusive land property rights could be exchanged. During the 19th century, the rise of the capitalism accelerated this movement towards more privatization, including that of culture and knowledge, inventing new tools such a copyrights and patents. Soon, we ended up believing that only two types of properties could exist, public vs. private, forgetting about this other kind of property that however used to be the norm for ages: ***the commons*** [Rifkin, 2014]. In its modern sense, the term "commons" refers to the broad set of resources that are our common heritage, the things that we use and exploit without tax, fee nor right of any kind, such as natural resources (*e.g.* land, atmosphere, oceans and forests), languages, cultures, knowledge, genetic heritage, etc. Conventionally, the commons also refer to the ways in which common resources are managed: these are various forms of sustainable co-ownership and self-governance, meant to regulate the access to, exploitation and sharing of common resources.

After some economists contended that commons were an archaic system unsuitable for the modern market economy [Hardin, 1968], political economist Elinor Ostrom[4] initiated a movement in the 80's to restore their nobility. She studied how some human societies had come up with long-term sustainable systems for the governance of natural resources. At the end of the 90's, Ostrom's work gained a renewed interest in a somewhat different context: the rapid deployment of the internet. Whereas she had observed commons at the level of local communities (villages, regions), the ubiquitous access to a digital infrastructure suddenly appeared as an ideal media to support worldwide communities that could collectively take care of new types of commons. Although Richard Stallman

[3] Enclosure movement: https://en.wikipedia.org/wiki/Inclosure_Acts ("Enclosure" is the modern spelling of "Inclosure")
[4] Elinor Ostrom: https://en.wikipedia.org/wiki/Elinor_Ostrom

did not explicitly refer to Ostrom's works, the emergence of the free software movement is clearly rooted in the same philosophical approach [Schweik & Kitsing, 2010; Broca & Coriat, 2015]. Since the early 2000's, the movement has grown rapidly, increasingly spurring the building of a "platform cooperativism" [Scholz, 2016] supporting a sharing economy where collaborative commons are produced, maintained and shared by user communities. The ***Open Data***[5] movement precisely lies in this trend. It inherits from the works of Ostrom, Stallman as well as political and philosophical movements grappling for the free circulation of knowledge, culture, and for the transparency of public institutions. In the next section, we further detail what it refers to.

2.2 Open Data

The Open Data movement advocates that some data should be considered as communal goods. As such, these knowledge commons[6] should be made freely available to the public, so that anyone could study, use, copy and redistribute them, in unmodified or modified form, without any restriction, copyright, patent or fee. Implicitly, the Web has become the natural platform to publish and share open data in digital formats.

Beyond the philosophical motivation, it is believed that making data freely available should drive innovation by allowing its repurposing, that is, the reuse of data in some new way, possibly beyond the field of application for which it was initially produced. This is the reason why, in his talk about "The next Web"[7], Tim Berners-Lee asked people to repeat with him the famous "Raw data now!". The idea is that, whatever the data, there will always be someone to find a clever way to exploit it and do something interesting with it.

The Open Knowledge Foundation[8] lists several topics of interest in which open data could have applications: scientific data, government accounts, financial markets, data from statistical offices, weather, pollution (*e.g.* air and rivers quality), transport (*e.g.* timetables, real-time traffic) and culture. In a more prospective view, some believe that new types of routinely available open data will drive profound social changes in a near future [Rifkin, 2014]. 3D-printing files shall empower a makers economy to print all sorts of objects, thereby helping to struggle against planned obsolescence with a localized production; crowed-sourced quantify-self data and medical records shall enable patient-driven medicine to advance research on rare diseases or discover new treatments; IoT data about energy consumption of buildings shall help enforce more efficient, large-scale energy policies, etc.

Although the idea of open data is not new, only recently has it been defined formally. The Open Definition[9] proposes that only two restrictions could apply to open data, (i) attribution: redistribution of the data should include authorship attribution and/or provenance information; and (ii) openness: redistribution of the data or derivative works should retain the same sharing conditions (a.k.a. share-

[5] https://en.wikipedia.org/wiki/Open_data

[6] The Knowledge Commons commonly include data from/about public institutions (e-government), scientific research, financial markets, transportation, pollution, arts, culture, etc.

[7] Tim Berners-Lee, Ted Talks: http://www.ted.com/talks/tim_berners_lee_on_the_next_web

[8] Open Knowledge Foundation: https://okfn.org/opendata/

[9] The Open Definition: http://opendefinition.org/od/2.1/en/

alike or copyleft). License frameworks such as the Open Data Commons[10] and Creative Commons[11] are typically meant to provide data producers with appropriate tools to distribute their work as open data. Notice that the Creative Commons "Non-Commercial" condition does not fit in the above definition. In this matter, Ball wrote a rich guide to licensing research data [Ball, 2014], yet most of the discussion applies to other types of data alike. A comparison of several licenses is provided, and the author points out the possible side effects of non-commercial and copyleft conditions.

The Open Data movement has gained an increasing popularity over the last two decades, fostered by the deployment of the Internet along with Web applications ideally suited to share data with a broad audience. The sheer fact that a worldwide network infrastructure is now available has stimulated and matured the idea of open government data, and although the open science movement largely predates the Internet, open science initiatives were pushed forward by the availability thereof. Below, we focus specifically on these two flagship domains of application of open data. Nevertheless, open data is much broader than this today. A great deal of internet services such as social networks or cartography services provide open access to some of their data, generally in the form of dedicated APIs, in order to allow for the development of third-party applications on top of their data.

2.2.1 Open Government Data

Providing access to government data meets a citizen's demand for access to data about publicly funded activities. It is expected to increase transparency, accountability and public awareness about government actions, about how resources are used and how public money is spent. It is expected to spur the involvement of the civil society in the development of new insights, thus enhancing public debate and fueling new outcomes to deal with societal issues such as health, education, public safety, environmental protection and governance. Open government data are also expected to foster innovation in the public and private sectors, supporting industries in the creation of new markets.

Although these expectations might be an idealized view, we must acknowledge an explosion of the interest in opening up public data. Many initiatives have been launched and many more are emerging every month, where cities, regions or governments are committed in the open publication of their data.

The UNData portal[12] was launched by the United Nations in 2008. It gathers and publishes a unique set of statistical data on a variety of topics, collected over 60 years by the UN member states and agencies. The European Union Open Data Portal[13], launched in 2012, provides a unified access to data stored on the Web sites of the different institutions and bodies of the union. The European Data Portal[14] is a single access point to the countless open datasets and data portals from cities, regional and national public bodies across European countries. It provides interesting counts of datasets per country, tag, topic or data format, as well as examples of applications exploiting these data. In addition,

[10] Open Data Commons: http://opendatacommons.org/

[11] Creative Commons: https://creativecommons.org/

[12] United Nations Data portal: http://data.un.org/

[13] EU Open Data Portal: http://data.europa.eu/euodp/: data from EU institutions and bodies (same as https://open-data.europa.eu/)

[14] European Data portal: http://www.europeandataportal.eu/ : data from member states

let us notice that, while these portals are an entry point to discover data sets, DataPortals.org is another entry point that inventories open data portals around the world.

In 2013, the G8 leaders signed the Open Data Charter [UK Gov. Cabinet Office, 2013] where the G8 member states set out to publish all government data openly by default, alongside principles to increase the quality, quantity and re-use of data. This charter coordinates the existing national initiatives such as the open data portal from France[15] or the United Kingdom[16], that provide counts of datasets per topic, data format, license, publishers, etc. The UK portal computes an interesting Openness Score referring to Linked Open Data star-rating system proposed by T. Berners Lee [Berners-Lee, 2006].

2.2.2 Open Science and Open Research Data

Open science refers to the will to share scientific knowledge and results within the society. The concept dates back to the 17th century with the creation and adoption of academic journals. Today, open science commonly encompasses the open access to scientific literature, the open peer-review process, the open access to research data, and the open-source licensing of software code since modern science largely hinges upon software to process and analyze research data. Several arguments motivate open science. Let us cite commonly admitted ones: from an ethical point of view, results of publicly funded research should be made available to the public; open science shall make research more transparent by allowing the reproducibility of research results, and should allow to engage the public in the scientific process; open science fosters a more collaborative and efficient science thus maximizing the social and economic benefits of research.

The open access to publicly funded scientific results in the form of peer-reviewed academic publications has been debated for several decades. The debate was significantly amplified by the generalization of electronic publications that questions the added value of commercial editors [Larivière et al., 2015]. Conversely, the open access to research data is a more recent concern [Murray-Rust, 2008]. Both are now considered alongside, regarding research data as an asset as valuable and strategic as academic publications. This growing awareness was marked by several milestones, from sheer letters of intention in the early 2000's to more concrete actions in recent years. Below we illustrate this evolution.

In 2003, many institutions worldwide signed the *Berlin Declaration on Open Access to Knowledge in the Sciences and Humanities* [Max Planck Gesellschaft, 2003] stating their intent to support the open access not only to publications but also to research data including all sorts of materials. The declaration already stipulates the rights to copy, use and redistribute data, provided that authorship is properly attributed. In 2004, members of the Organisation for Economic Co-operation and Development (OECD) signed a declaration targeting more specifically research data. It states that all publicly funded archive data should be made publicly available. In 2013, the G8 science ministers and national science academies published a statement proposing new principles of collaboration with respect to research infrastructures and open access to scientific results as well as research data [G8 Science Ministers,

[15] French open data portal: https://data.gouv.fr
[16] UK open data portal: https://data.gov.uk/data/search

2013]. This open access should be enabled "to the greatest extent and with the fewest constraints possible". In particular, the statement emphasizes that successful adoption of these principles should be underpinned by an appropriate recognition of researchers fulfilling them. The Horizon 2020 European program puts a specific stress on the open access to publications and research data[17] through an Open Research Data pilot [European Commission, 2016]. Project proposals are requested to include a data management plan (DMP) that details the life cycle of the data generated by the project. The DMP must address several questions such as what data will be produced during the project, using which formats/standards, how data will be curated, openly shared and sustainably archived. From 2017 on, the pilot applies to all the thematic areas of H2020. The OpenAIRE[18] H2020 project aims to implement the EU's open access policy. It intends to become the access point to all European funded research outcomes through the coordination of a network of open repositories and archives. In particular, the Zenodo[19] research data repository is a product of OpenAIRE.

In addition to the philosophical aspects of open data and the political will to set out for them, some works have studied the type of research data that can be shared. In the domain of neuroscience for instance, Fergusson et al. point out that major data sharing initiatives target big, organized, curated data sets. But routinely sharing "individual, small-scale studies (…) known as long-tail data" is barely tackled [Ferguson et al., 2014]. The long tail typically consists of data considered as ancillary to published works: results from pilot studies or failed experiments, figures, posters, reviews, slides, etc. Individually, they may have little value but collectively, it is believed that they would entail great added value. Fergusson et al. also investigate the lack of incentive for scientists to share their data, as many consider sharing more as a threat and a waste of time than an opportunity. They conclude by reminding the pressing need for a credit attribution system such that researchers would be rewarded for the publication of scientific articles and research data alike. The emergence of data papers and data journals is an encouraging step toward this end. Data papers provide metadata about published data sets, they are peer-reviewed and thereby allow to credit researchers who make data available with a citable persistent data identifier (such as a DOI).

2.3 Linked Data and the Web of Data

The initiatives cited in the previous section are all geared toward the same goal: making all sorts of data available to the public under a license that respects the open data philosophy. Yet, the sheer publication of open data on the Web may not be sufficient to make it truly useful. The real power of open data lies in the fact that connecting related pieces of data increases their respective value. To create added value by consuming and mashing up open data, a program should be able to browse through a data set and automatically discover related data sets, very much like a human follows HTML links from one Web site to another one that somehow pertains to related topics.

[17] EU Open Science/Access: https://ec.europa.eu/programmes/horizon2020/en/h2020-section/open-science-open-access
[18] OpenAIRE project: http://www.openaire.eu/
[19] Zenodo open repository: https://zenodo.org/

Many open data sets available on the Web today are provided through specialized Web APIs, each coming with its own authentication rules, access syntaxes and vocabularies. A study[20] reports the creation of more than 4000 APIs from 2006 to 2011, and ProgrammableWeb.com currently reports over 16,000 such APIs. This situation has led to the development of countless applications specialized in the combination of data from well-identified data sources, thus ensuring structural and semantic interoperability between those sources. Whereas this approach is effective, it bears significant impediments, commented in [Heath & Bizer, 2011]:

(i) High maintenance costs must be supported when the integration of additional sources is required, or to follow up on API changes.
(ii) Although several data sets may refer to the same logical object, they would typically use local identifiers. The reconciliation of local identifiers throughout sources has to be done on a case-by-case basis by applications.
(iii) Logical objects do not truly exist on the Web; they are only accessible by means of Web APIs. Consequently, there is no standard mechanism to browse through data sources and discover related resources.
(iv) Let us add that Web APIs restrict the query forms to a set of predefined set of queries, thus lacking query flexibility.

For decades, data integration principles have relied on specific applications to connect data items from different databases within unified data models. This is still the approach implied by Web APIs today: each data provider designs its own API independently, overlooking interoperability issues and delegating them to the data consumer. However, when we consider the Web-scale integration of countless data sources, this model does no longer fit.

To address this challenge, the **Linked Data** principles, formulated by Tim Berners-Lee in a note on Web architecture [Berners-Lee, 2006], recommend best practices for publishing data on the Web while specifying the existence and meaning of connections between data items. These principles rely on existing standard Web building blocks: URI and HTTP. They advocate (i) the use of URIs to name things (may they be digital artefacts, documents, physical objects, abstract concepts, people, etc.); (ii) the use of HTTP as the transport layer so that HTTP URIs can be looked up[21]; (iii) the use of standards: the machine-readable Resource Description Framework (RDF) [Cyganiak et al., 2014] to represent information and SPARQL to query it [Harris & Seaborne, 2013, 2008]; and (iv) the inclusion of RDF typed links between data sources so that connected information can be discovered at run time by following the links (similar to the way HTML anchors are used to link documents from different Web sites). Heath and Bizer described and commented in details those principles [Heath & Bizer, 2011].

In the last 10 years, a growing number of organizations and services have started to publish their data in compliance with the Linked Data principles. The emerging result is referred to as the **Web of Data** [Bizer, 2009]. This marks the extension of a Web of documents woven by HTML hyperlinks meant for humans, towards a Web of data woven by shared URIs meant for machines. Later in 2010, Berners-Lee

[20] Evolution of Open Data Web APIs from 2006 to 2011: http://visual.ly/open-data-movement

[21] These principles have been extended so that resources identified by URIs can be created, modified or deleted using HTTP methods (see Linked Data Platform [Speicher et al., 2015].

amended his note on Linked Data to define **Linked Open Data** as "Linked Data which is released under an open license". This is illustrated by the five-star rating system for Linked Open Data[22] that he proposed, where each rating adds a property to the properties of the previous rating:

★	Data is made available on the Web in any format but with an open license.
★★	Data is available in a machine-readable, structured format.
★★★	The data format is non-proprietary.
★★★★	Open standards from the W3C are used: URIs to identify things, RDF to describe data and SPARQL to query them.
★★★★★	Data sets are linked to other data sets to provide context.

An interesting property of the Linked Open Data principles is that they need not be applied all at once. Instead, they represent a virtuous goal that can be reached progressively, including each principle step by step from the sheer publication on the Web of data in any form, until the full 5-star Linked Open Data compliance.

Driven by these principles, the **Linking Open Data** community project[23] aims to bootstrap the Web of Data by identifying open data sets and fostering their publication in compliance with the Linked Open Data principles. The goal is to solve the semantic Web chicken-and-egg dilemma, stating that a critical mass of machine-readable data must be available for novel mash-up applications to arise. At the time of writing, the latest diagram of the Linking Open Data cloud was published in February 2017[24] (see Figure 2). It lists 1139 interlinked datasets in very diverse topics such as life sciences, geographical data, social networks, media, government data or general-purpose sources like DBpedia. Some of these datasets are actually themselves curated collections of other datasets, such as the Bio2RDF project that gathers over 30 linked life-science data sets about proteins, genes, drugs, scientific literature, taxonomical reference, etc.

This diagram is generated from metadata automatically collected from the DataHub.io portal, but unfortunately, authors acknowledge that it is now largely outdated. Besides, it lists datasets that were explicitly brought to the attention of the project, and cannot automatically discover open data sets published and maintained by other initiatives. For instance, the cloud references data portals from the American and British governments, but other countries are not included although most data portals from *e.g.* the European Union and its member states now do provide RDF data sets in addition to at least a catalog of data sets as Linked Data. Consequently, it is difficult to figure out the precise extent of the Web of Data today. Nevertheless, the LODStats project computes statistics about a subset of the Web of Data consisting of the Linking Open Data cloud and datasets from the American data.gov and European publicdata.eu[25] portals [Ermilov et al., 2016]. The statistics provide insights in this fragment of the Web of Data analyzed, with regard to *e.g.* the data sets, the variety and frequency of

[22] Five-Star Linked Open Data: http://5stardata.info/en/

[23] Linking Open Data community project: http://linkeddata.org/

[24] A browseable version of the diagram is available at: http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/

[25] The PulicData.eu portal was developed by the Open Knowledge Foundation and precedes the EU Open Data Portal: https://data.europa.eu/euodp/en/data

vocabularies being used, the number of links between data sets, the top-used properties, classes etc. Finally, let us recall that even though more and more data sets are natively published in RDF, they only account for a fraction of the open data available today. For instance, only 2% of the 587,000+ datasets on the European Data Portal are provided in RDF, while 22% are in the CSV, HTML, PDF or Excel formats. On the EU Open Data Portal, the ratio is of 2% in RDF to 81%, and the American data.gov portal scores better with over 6% of the datasets in RDF to 69% in CSV, HTML, PDF or Excel[26].

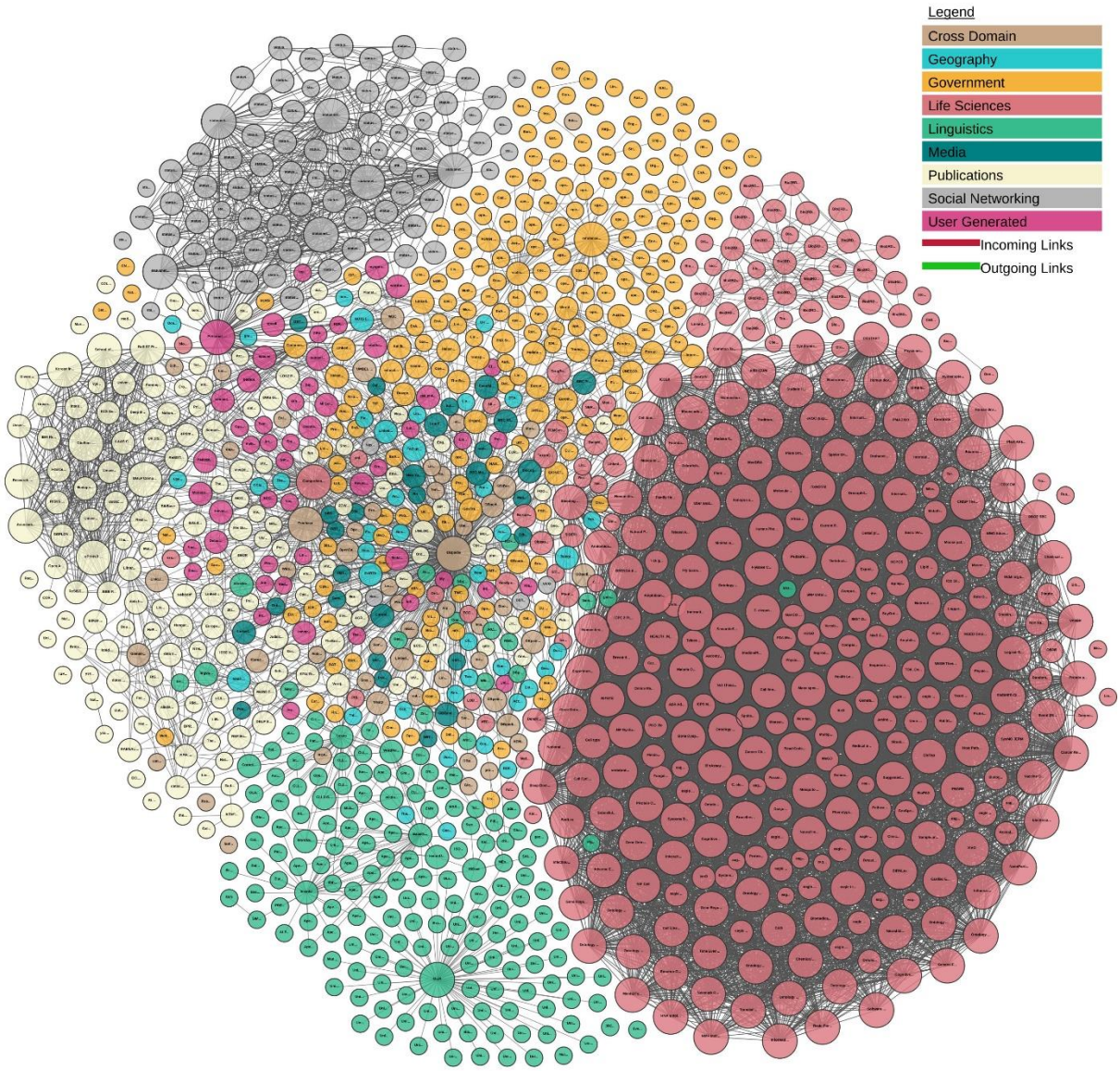


Figure 2: Linking Open Data cloud diagram, 20th Feb. 2017[27]

In the scientific community, some works try to harness the joined power of Open Science and Linked Data to go one step further towards more transparent, reproducible and transdisciplinary research. **Linked Open Science**, or Linked Science for short [Kauppinen & Espindola, 2011], is an emerging

[26] Figures based on the per-format statistics available on each of these portals in November 2016.

[27] Linking Open Data cloud diagram 2017, by Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net/

approach undertaking the linking of all scientific assets into **executable papers**[28]. The concept of executable paper advocates the adaptation of traditional journal articles to the needs of data intensive science. Concretely, an executable paper shall interlink articles, data, metadata, methods, software etc., thus delivering a validated, citable, tractable, and executable experimental context. To achieve this, Linked Open Science builds on several key components: (i) Linked Data to annotate and/or represent articles, data and metadata; (ii) Open Source and Web-based environments to provide software tools and methods thereby making it easy to reproduce experiments; (iii) Cloud-computing environments to make it easy to repeat CPU and space intensive tasks; (iv) Creative Commons licensing to provide a legal framework in which all scientific assets can be reused. Note that the latter condition may not always be appropriate: some licenses are better suited to deal with data and metadata *e.g.* Open Data Commons. LinkedScience.org is a community-driven project started in 2011, meant to showcase what Linked Open Science is about in practice. Not only does the project aim at reproducible science, but it also puts a specific stress on the need for education and dissemination of scientific results. Through different types of events, LinkedScience.org spurs Linked Science among scientific communities, and promotes tools and workflows that could facilitate the practice of Linked Science.

Other initiatives address similar challenges although not necessarily under the term Linked Science. This is the case of four major projects funded by the European Union FP7 program: BioMedBridges, CRIPS, DASISH and ENVRI are clusters of research infrastructures in biomedical sciences, physics, social sciences and humanities, and environmental sciences respectively. They have come together to identify the common challenges across scientific disciplines with respect to data management, sharing and integration [Field et al., 2013]. They have drawn a list of topics of interest covering notably traceable and citable research objects (data, software, user), semantic interoperability (interlinked vocabularies and ontologies, context and provenance metadata), and data processing services (description, composition, discovery, marketplace). For all of these topics, recommendations extensively rely on the concepts of Linked Data, and linked science is implied in the idea of linking together all assets needed to reproduce an experiment.

2.4 NoSQL Databases

2.4.1 A Short History

At the end of the 90's, relational databases (RDB) were the dominant type of database and it almost seemed obvious that they were the one-fits-all solution to any database needs. They were hardly challenged by a few other types of databases developed over time, for either generic purposes or niche domains, such as XML databases (notably used in edition), object-oriented or directory-based databases. The fact is that decades of research and engineering had built strong theoretical models providing RDB systems with properties that are desirable for most applications: atomicity, consistency, isolation, durability (*aka.* ACID properties), scalability, security and performance optimizations.

The advent of the Web 2.0 and the emergence of the cloud computing in the years 2000 rapidly changed the situation. At that time, major Web companies (*e.g.* e-commerce Web sites, social

[28] Executable Papers: http://www.executablepapers.com/about-challenge.html

networks) started to exhibit unprecedented needs that RDBs could hardly fulfill. Not only their new applications were meant to manage huge volumes of data, but they also required more flexible database schemas to keep up with rapidly changing needs, high throughput, high availability, and flexible infrastructures to scale up or down quickly. Parallel RDBs are able to deal with very large data sets but they have major drawbacks: they lack flexibility (scaling up is much more complicated than simply provisioning new hardware), the cost associated with scaling is very high, and the distribution over geographically distant sites significantly impairs performances. Therefore, taking advantage of the emerging cloud-computing technologies, engineers designed alternative types of distributed databases able to scale horizontally on elastic infrastructures consisting of thousands of cheap commodity hardware servers. These were called the "NoSQL" family of databases, a term first coined by Carlo Strozzi in 1998 but twisted and re-appropriated during the 2000's to name the emerging family of systems.

Commons examples of the NoSQL pioneers include BigTable (Google), Dynamo (Amazon), Voldemort (LinkedIn), Cassandra (Facebook), MongoDB and CouchDB. Some of them inspired other projects such as HyperTable and HBase derived from BigTable.

2.4.2 Architectures

Today, the NoSQL family spans a broad range of systems with usually admitted commonalities: capability to scale horizontally, replication/distribution of data on multiple servers, flexible schema or schema-less, ability to reach high throughput along with a high availability. As a tradeoff to scalability, high throughput and availability, NoSQL systems relax traditional ACID properties, adopting a weak consistency model that shifts the consistency management burden from the database level to the application level. They favor a weaker model named BASE [Pritchett, 2008] - a nod to the ACID acronym - standing for "Basically Available, Soft State, Eventual Consistency". An application following the BASE model is available at all times (basically available), it may not always be in a consistent state (soft-state) but consistency shall be restored at some time (eventual consistency). It builds upon the CAP Theorem [Brewer, 2000] which states that a distributed system cannot simultaneously enforce the three following properties: **Consistency** (after an update operation, all readers see the same up-to-date data), **Availability** (the system continues to answer queries even in case hardware/software failures occur), tolerance to network **Partitioning** (the system continues to operate in the case of partitions created by network failures). NoSQL systems generally favor availability and partition tolerance over consistency, *i.e.* in the presence of a network partition or hardware failure, the system will always return an answer although it cannot guarantee that this answer is the most up to date. But under no circumstance shall it time out or return an error.

Conventionally, architectures of NoSQL systems fall into four coarse categories: key-value or tuple store, document store, column store (aka. column family store, wide/extensible column store), and graph database. This categorization is not strict and some systems are considered multi-model. Questionably, XML and object-oriented databases are sometimes considered as NoSQL categories because they do not comply with the relational model[29]. This is however confusing as they largely pre-existed the NoSQL movement and do not specifically focus on high throughput and availability. The

[29] This is notably the case of http://nosql-database.org/

different architectures were abundantly described and compared in the literature. Below we briefly describe salient features of the four aforementioned types of NoSQL databases, and we refer the interested reader to the literature for further details [Gajendran, 2013; Hecht & Jablonski, 2011].

Key-value stores. Key-value stores are schema-free maps/dictionaries of entries that are stored and retrieved by a unique key. Search can be achieved using keys but values are opaque to the system. Hence, values are not indexed and no advanced querying features are available. They are often implemented as distributed in-memory databases, and optimized for very fast retrieval. They are typically designed to fulfil the role of highly available cache for frequently accessed information such as user profiles or session ids, or as a front-end caches for more time intensive query systems. Key-value stores provide very simple query capabilities, amounting essentially to storing/updating or retrieving a value by its key.

Some prominent key-value stores include DynamoDB[30] (Amazon), Voldemort[31] (an open-source implementation of Dynamo, used notably by LinkedIn), Redis[32] and Riak[33].

Document stores. Document stores are an evolution of key-value stores where values are no longer opaque to the system: they are generally JSON or JSON-like documents with a mandatory unique identifier. Document stores generally allow the definition of indexes on document attributes. This comes with much richer querying capabilities than key-value stores, in the form of proprietary query languages. They are schema-less in that JSON documents can have any arbitrary form; the application is responsible for managing the schema, if any. Document stores are optimized for fast storage and retrieval of very large collections of documents. They generally do not support joins, as a result documents comply with denormalized data models where redundancy is common.

MongoDB[34] and Apache CouchDB[35] are the most well-known and commonly cited document stores.

Extensible Column stores. Most column stores were inspired by the Google's pioneer Bigtable project [Chang et al., 2008]. They follow a regular tabular data model, and can support very large tables comprising thousands of columns. Distribution on many nodes is achieved using different policies involving both horizontal and vertical partitioning. Unlike traditional relational databases, the schema is flexible in column stores: a record (a table row) can have any number of columns, and columns can be added on the fly in a specific row. For this reason, rows can be seen more accurately as key-value maps. Yet, these systems are totally not schema-less though. Groups (or families) of columns can only be defined to allow for more efficient distribution/sharding, but those groups must be defined statically. Column stores also generally natively support versioning of values. Similar to document stores, relationships (joins) are not supported in such databases; hence, they must be handled in the application. Column stores general come with an SQL-like query language, with varying limitations compared to SQL.

[30] DynamoDB: https://aws.amazon.com/dynamodb/
[31] Voledmort: http://www.project-voldemort.com/voldemort/
[32] Redis: https://redis.io/
[33] Riak: http://basho.com/products/riak-kv/
[34] MongoDB: https://www.mongodb.com/
[35] CouchDB: http://couchdb.apache.org/

Most popular extensible stores are Google's Bigtable, Apache Cassandra[36], Hypertable[37] and Apache HBase[38].

Graph stores. Graph stores are designed to manage nodes linked labeled, oriented edges. A set of key-value pairs can be attached to each node. Unlike the three aforementioned types of NoSQL databases that handle collections of independent documents, graph stores are focused on representing relationships between entities, thus being appropriate to represent highly connected graphs. Some of them implement the RDF model although but many follow different models. They come with rich query capabilities, allowing to query or navigate through a graph.

Among the well-known graph stores, we can cite Neo4j[39], OrientDB[40], Allegrograph[41], GraphDB[42].

Finally, let us add that this strict classification can now seem quite artificial. Indeed, the frontiers between these four types are increasingly porous as more and more systems adopt a multi-model approach. This is the case, for instance, of ArrangoDB[43], which is a graph, key-value and document store all together, or Couchbase Server[44] which is both a key-value and document store.

[36] Cassandra: https://cassandra.apache.org/
[37] HyperTable: http://www.hypertable.org/
[38] HBase: http://www.hypertable.org/
[39] Neo4J : https://neo4j.com/
[40] OrientDB: http://orientdb.com/
[41] Allegrograph: http://franz.com/agraph/allegrograph/
[42] GraphDB : http://graphdb.ontotext.com/graphdb/
[43] ArrangoDB: https://www.arangodb.com/
[44] Couchbase Server: https://www.couchbase.com/

Chapter 3. State of the Art: from Data Integration to Ontology-Based Data Access

Data Integration refers to the techniques involved in combining data residing at different locations into a common, integrated view. This unified view is modelled by the global schema (or mediated schema) and provides a reconciled view of the data sources. The domain rose during the 90's as the need for decision-support systems based on analytical reports lead to the development of a family of products called data warehouses. Since then, these systems have been very successful in integrating the manifold databases of an enterprise's information system along a controlled, unified schema.

Since the early 2000's however, new needs have emerged. The availability of distributed, heterogeneous resources has led to consider broader, looser data integration approaches where independent data sources can participate in virtual data federations. While data warehouses are rigid repositories controlled within the premises of a single company, these new needs must accommodate the opportunistic addition of new data sources from independent institutions. Furthermore, data semantics is often poorly captured in database schemas. To some extent, implicit semantics can be figured out, *e.g.*, from integrity constraints or usual database design patterns, but additional semantics is frequently encoded in the application exploiting a data source. Moreover, database schemas are often fine-tuned for performance reasons, which results in mixing up data semantics with technical concerns. Therefore, Web-scale data integration techniques require the ability to capture, agree on and share common conceptual formalizations in an explicit and machine-readable manner. This is conventionally achieved by means of controlled vocabularies, thesauri or ontologies.

In particular, biologists and neuroscientists have realized very early that data may gain value by being used beyond the purpose for which they were initially acquired [Martone et al., 2004; Akil et al., 2011]. Thus, to enable reasoning throughout different scales, from molecule to brain functions and from cell to organism, many projects have investigated how to achieve semantic reconciliation of heterogeneous, distributed data sources, *e.g.* [Gardner et al., 2008; Keator et al., 2008; Gibaud et al., 2011].

In this chapter, we first review data integration principles, specifically the types of mappings used to reconcile data sources with a global schema (section 3.1). Then, we focus on a family of data integration approaches that build upon ontologies as the global schema (section 3.2). Whatever the approach, mappings are always a keystone in data integration systems. Therefore, since our focus is on data integration relying on semantic Web technologies, we review previous works dedicated to the translation of existing data formats and databases into RDF (section 3.3).

3.1 Data Integration Principles

Conventionally, a data integration system I is denoted by the tuple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$: the global schema $\mathcal{G}$ is used to represent the unified view, the data sources $\mathcal{S}$ are represented by the set of local schemas $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, and the mapping $\mathcal{M}$ specifies correspondences between concepts of the local schemas and concepts of the global schema. Depending on the modeling languages used to define the global and local schemas, concepts may be ontological classes, relational tables, objects etc.

A data integration system answers queries posed in terms of the global schema $\mathcal{G}$ by reformulating them into appropriate queries over the data sources $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$, utilizing the information captured in the mapping $\mathcal{M}$. Two main approaches have been proposed with regards to the way a mapping is expressed: in the Global-as-View approach, the global schema is expressed in terms of queries (or views) over the local schemas; conversely, in the Local-as-View approach, the local schemas are expressed in terms of queries (or views) over the global schema. These approaches, as well as the in-between Global-and-Local-as-View, have been abundantly described in the literature [Doan et al., 2012; Lenzerini, 2002]. In this section, we remind their main characteristics and discuss their relevance in different contexts.

3.1.1 Global-as-View

Global-as-View (GAV) is a quite natural approach to specifying mappings, where each concept of the global schema is defined in terms of a query over the data sources. For each concept $g \in \mathcal{G}$, the mapping $\mathcal{M}$ specifies expressions of the form

$g \supseteq Q(\mathcal{S})$ under the open world assumption, or

$g = Q(\mathcal{S})$ under the closed world assumption,

where $Q(\mathcal{S})$ is a query over the local schemas $\mathcal{S}_1, \ldots \mathcal{S}_n$. Informally, these expressions tell the data integration system how to use the data sources to yield data corresponding to each concept of the global schema. In the open world assumption, the expression $g \supseteq Q(\mathcal{S})$ illustrates that query $Q(\mathcal{S})$ may participate in the production of instances of concept g, but it may not be sufficient to yield all instances of g.

The main advantage of the GAV mapping is its conceptual simplicity: to answer a query over the global schema, a system must simply unfold the query, that is, substitute the concepts of the global schema with the corresponding definitions. Furthermore, the GAV approach easily supports evolutions of the global schema. Indeed, since the mapping of each concept is specified independently of the others, adding a new concept to the global schema, or changing its definition, only involves writing or updating the mappings for that specific concept, but it does not impact existing mappings.

On the other hand, GAV has several drawbacks:

- Writing the mapping of a concept g of the global schema requires knowing all local schemas, in order to identify the sources that are relevant for g.
- Adding a new source may require updating the mappings of several concepts in which the new source shall contribute.
- GAV is not fault-tolerant: the unavailability of one data source is sufficient to prevent the system from working, since the concepts of the global schema that involve that source cannot be processed.

Consequently, it is generally admitted that the GAV approach is appropriate for a limited number of data sources, where adding or removing a data source is uncommon, and where the local schemas hardly evolve in time.

Finally, a common argument against the GAV approach is that adding a new source may jeopardize the system by inducing changes in the global schema. This is only partially true and closely depends on the context. Let us illustrate this with two examples:

(1) An enterprise has created a global data model to integrate various databases of its information system, and it now wishes to integrate an additional data source. If that new data source concerns a domain (HR, billing, customer relation etc.) that is not or only partially covered by the global schema, then indeed, the global schema must be updated. This entails the risk that many mappings have to be updated too.

(2) A global schema defines a domain of knowledge on which relevant data sources are aligned using the GAV approach. Whenever a source is integrated, the schema remains the same: only the relevant concepts of the source are mapped to the global schema, others are simply ignored. This was *e.g.* the approach of the NeuroLOG project [Michel et al., 2010] for the federation of neuroimaging data sources. A fixed global schema was defined, covering various concepts of neuroimaging (image types, acquisition modalities and protocols, etc.). Databases from several clinical centers were progressively integrated into the federation using GAV mappings in a best effort manner: data that could not be represented in the global schema were simply not mapped, but this did not entail any change in the global schema.

While the first example is typical of the integration of enterprise legacy databases, the second example illustrates a more recent trend of data integration systems focused on an open world where data is opportunistically aligned on an existing domain formalization.

3.1.2 Local-as-View

The second approach is less intuitive than the GAV. The Local-as-View (LAV) defines each concept of the local schemas in terms of a query (view) over the global schema. For each concept s of a local schema $\mathcal{S}_i$, the mapping $\mathcal{M}$ specifies expressions of the form

$s \subseteq Q(\mathcal{G})$ under the open world assumption, or

$s = Q(\mathcal{G})$ under the closed world assumption,

where $Q(\mathcal{G})$ is a query over the global schema.

The main advantage of the LAV mapping is that each data source is described independently of the others, which allows the approach to scale easily to any number of sources (unlike the GAV approach). Besides, the independence of each data source makes the system resilient to single data source failures: the unavailability of one data source does not prevent the system from working, possibly in an impaired manner, since the concepts of the global schema can still be yielded by other data sources.

The LAV approach has two major drawbacks:

- Adding a new concept or updating a concept in the global schema may require updating the mappings of several data sources. This may become impossible in practice when a high number of sources is considered.
- The flexibility gained in terms of scalability is paid back in term of computational complexity of query processing. The problem amounts to a more general class of problems called "query answering using views" [Doan et al., 2012]. In the LAV context, the query reformulation problem is NP-complete for conjunctive queries, and in the worst case, the number of rewritings of a query may be exponential in the size of the query and the number of views. This is notably critical when some concepts of the global schema are shared by many views (*i.e.* when many sources provide data relevant for the same global concept). Consequently, the LAV approach is more appropriate with vertically partitioned sources, *i.e.* each concept of the global schema is preferably provided by one or only a small subset of the sources.

In Table 1, we summarize the benefits and concerns of the GAV and LAV approaches.

Table 1: Benefits and concerns of the GAV and LAV approaches

	Global-as-View	Local-as-View
Evolution of the global schema	✓	×
Evolution of the data sources (add/remove, change the local schema)	×	✓
Scaling to many data sources	×	✓
Tolerance to data source unavailability	×	✓
Query answering	Simple (unfolding)	Difficult (NP-complete)
Query non-vertically partitioned data sources	✓	×

3.1.3 Global-and-Local-as-View

Let us notice a third approach that combines the GAV and LAV approaches. In the Global-and-Local-as-View (GLAV) approach [Friedman et al., 1999], the mapping $\mathcal{M}$ specifies expressions of the form

$Q(\mathcal{S}) \subseteq Q(\mathcal{G})$ under the open world assumption, or

$Q(\mathcal{S}) = Q(\mathcal{G})$ under the closed world assumption,

where $Q(\mathcal{G})$ is a conjunctive query over the global schema and $Q(\mathcal{S})$ is a conjunctive query over the local schemas.

Informally, query answering combines the two techniques. First, the query is rewritten using the views over the global schema (the LAV part). Then, each view over the global schema is unfolded using the queries over the local schemas (the GAV part). It is shown that the GLAV query processing cost is the same as the LAV query processing cost in the worst case.

3.1.4 RDF-Based Data Integration Systems

In the RDF-based data integration context, several systems have been proposed in the form of federated SPARQL query engines [Schwarte et al., 2011; Görlitz & Staab, 2011; Corby et al., 2012; Macina et al., 2016] or Linked Data fragments [Verborgh et al., 2016]. These systems cannot clearly be classified as either GAV or LAV approaches. Indeed, their point is to produce efficient query plans over distributed data sources supporting SPARQL, but there is no schema mediation: the global schema consists of the set of ontologies used in the SPARQL query, and data sources are selected according to whether they can produce triples referring to those ontologies.

Interestingly enough, only one LAV-based approach for the Semantic Web has been proposed so far. SemLAV tackles the federated querying of multiple data sources from the Web of Data [Montoya et al., 2013]. To avoid the NP-complete problem of LAV query rewriting, SemLAV formalizes the problem of selecting and prioritizing the most relevant views to maximize the number of covered rewritings: it selects and sorts out the sources that are more likely to produce answers, but the different possible rewritings are never actually computed. Instead, most relevant views (those that cover most rewritings) are materialized into an aggregation graph on which the SPARQL query is evaluated. Thus, SemLAV can start returning results very quickly (after only a handful of views are materialized). Progressively, more views are materialized, until enough results have been retrieved or a time limit has expired. In addition, a strong interest of SemLAV is that it supports SPARQL queries that are not limited to conjunctive queries, unlike in traditional LAV approaches, but encompasses all SPARQL expressiveness.

3.2 Ontology-Based Data Access

In the previous section, we have sketched how Data Integration systems can map data sources to a unified view, the global schema. Yet, how this global schema is defined is not addressed. Ontology-Based Data Access (OBDA) is a domain of Data Integration that advocates the utilization of ontologies as the formal conceptual layer: an ontology represents the domain of data stored in a data source, and the mapping describes the relationships between the ontology and the data source. Then, a query is specified in terms of this ontology and the OBDA system translates the query into queries over the data source.

Principles of OBDA have been extensively studied in the context of relational databases (RBD) [Poggi et al., 2008]. Let us first remind what makes querying a relational database or an ontology so different. In RDBs, the schema is used at design time to describe constraints. It is not used during query processing since data is assumed to respect this schema by design (we refer to this property as the closed-world assumption). Consequently, query answering in RDBs simply amounts to query evaluation (*e.g.* concept membership checking) which is computationally cheap. On the other hand,

an ontology defines constraints on the data, but data may be incomplete or inconsistent with respect to these constraints (this is the open-world assumption). Therefore, unlike RDBs, it is necessary to take the constraints into account during query answering, to overcome incompleteness or inconsistency. Hence, query answering in an ontology-based system requires run-time logical inference, which is computationally costly. In the following, we describe the different options to tackle this problem.

Conventionally, an ontology $\mathcal{O}$ is denoted by a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ where $\mathcal{T}$ represents the intentional level information (terminological box, or T-box) and consists of axioms with respect to concepts (classes) and roles (predicates); $\mathcal{A}$ is the set of facts about individuals (assertion box, or A-box), it consists of concept and role membership assertions. Answering a query Q posed in terms of an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ can be achieved according to two approaches depicted in Figure 3:

- Perform logical inference on the A-box, considered incomplete due to open-world assumption, using the T-box (arrow 1). This problem is considered hard as it depends on the size of the A-box which is generally very large.
- Perform logical inference on the query Q using the T-box (arrow 2). This option scales much better since the reasoning is performed on the query whose size is typically very small compared to the A-box. Then, it is shown that the resulting query Q' can be evaluated against the A-box that is now considered under the closed-world assumption (arrow 3)[45].

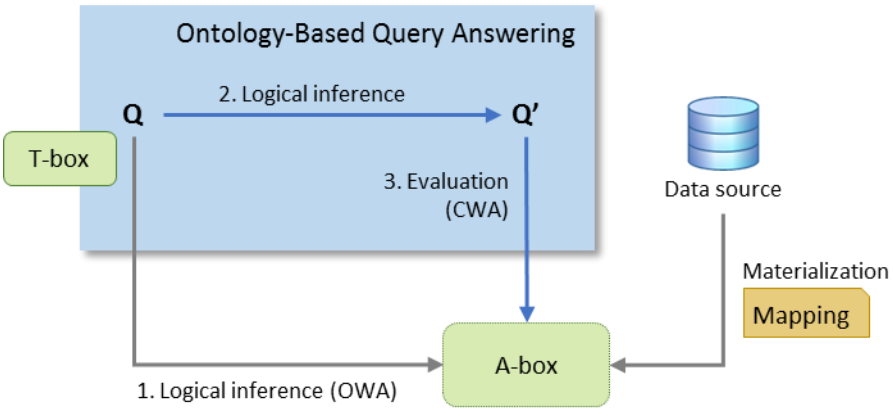


Figure 3: Ontology-Based Query Answering

Yet, in Ontology-Based Query Answering (OBQA), the rewritten query Q' is still posed over the ontology, not over the data source. Thus, it requires the A-box to be materialized in the first place, by translating data from the source into ontological assertions of the A-box with respect to a mapping (arrow *Materialization* in Figure 3). In practice, materializing the A-box is hardly possible when we consider large data sets. Therefore, the OBDA approach exploits Data Integration principles so that the data remain at the source. This is depicted in Figure 4: the mapping is used to reformulate the query Q' into a query Q'' over the data source (arrow 3). Since in this case we are concerned with only one data source, the mapping generally complies with the Global-as-View approach (although the Local-as-View may be possible). Then, query Q'' is simply evaluated against the data source under the closed-

[45] This result holds under the conditions that O is a DL-Lite satisfiable ontology and Q is a conjunctive query or a union of conjunctive queries.

world assumption (arrow 4). This yields the same results as if Q' was evaluated on the A-box which is no longer materialized but virtual (denoted by the dotted arrow and box).

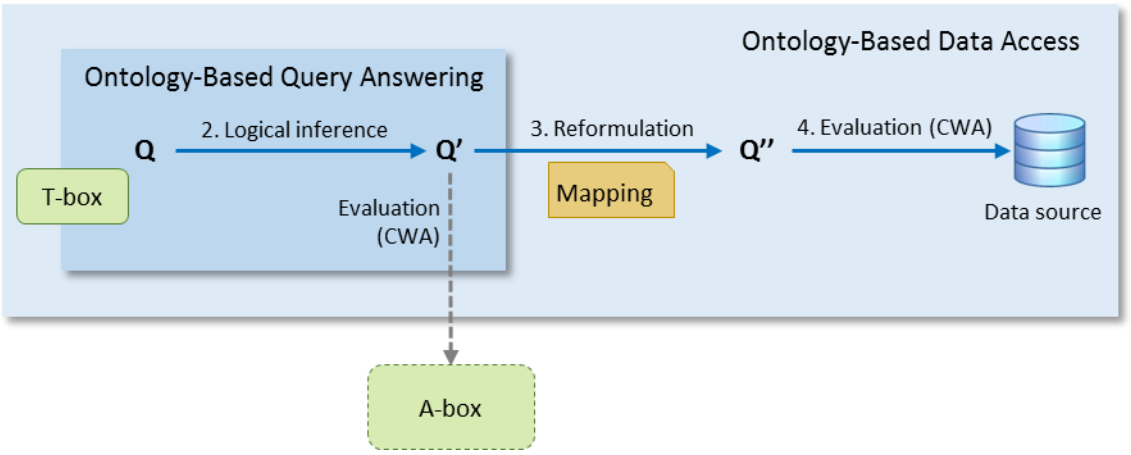


Figure 4: Ontology-Based Query Answering vs. Ontology-Based Data Access

OBDA in RDF-based data integration systems. In the context of RDF-based data integration systems, the OBDA principles have been demonstrated in the Ontop project [Rodríguez-Muro et al., 2013], depicted in Figure 5. In this context,

- the ontology is expressed in the OWL2 QL profile;
- Q is a SPARQL conjunctive query posed in terms of the ontology;
- the rewritten SPARQL query Q' is a union of conjunctive queries;
- the mapping is represented by a set of Datalog rewriting rules;
- the data source is a relational database, hence the reformulated query Q'' is written in SQL.

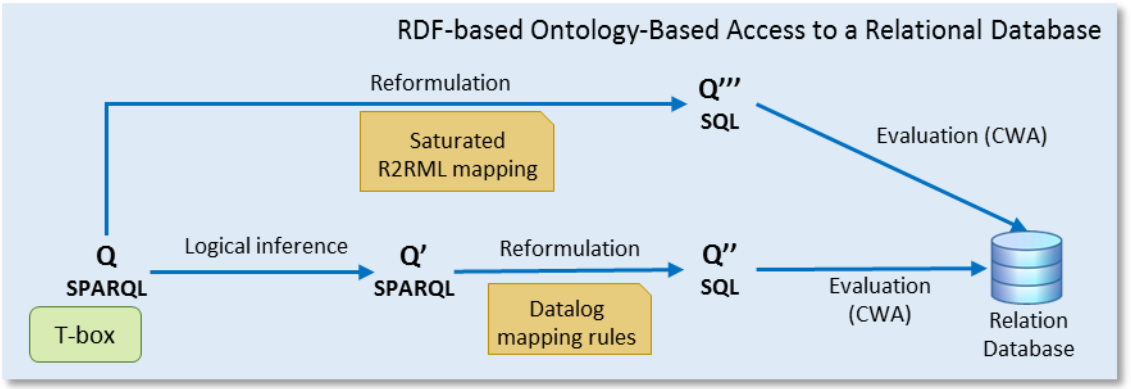


Figure 5: RDF-based Ontology-Based Data Access to a Relational Database

An alternative to this approach proposes to reason, not on the query itself, but on the mapping between the data source and the ontology [Sequeda et al., 2014]. Logical inference embeds ontological entailments in the Global-as-View mapping $\mathcal{M}$ (R2RML in this case) giving rise to the saturated mapping $\mathcal{M}$*. For instance, if a mapping yields triple "<someUri> rdf:type _:A" and the ontology specifies that A is a sub-class of B, then a new mapping is created that yields triple "<someUri> rdf:type _:B". Authors show that reformulating a SPARQL query Q using the saturated mapping $\mathcal{M}$* produces an SQL query (Q''' in Figure 5) that can be evaluated against the database under the closed-world assumption. Both reformulated queries, Q'' and Q''' are equivalent.

OBDA applied to non-relational databases. Beyond relational databases, some works have proposed methods to apply OBDA principles to some NoSQL databases. Mugnier et al consider the key-value store subset of NoSQL databases [Mugnier et al., 2016]. Considering two types of queries, the `get(k)` query that retrieves the value associated with a key-path k and `check(k)` that verifies the existence of a value associated with a key-path k, they propose the NO-RL rule language that can express lightweight ontologies to be applied on key-value stores. Based on the formal semantics of how NO-RL rules can enrich records of a key-value store, the authors propose an algorithm to reformulate a query under a NO-RL ontology.

Botoeva et al. propose to apply the OBDA principles to query the MongoDB document store [Botoeva et al., 2016a]. They first "flatten" MongoDB documents to obtain a tabular view, then they use the Ontop OBDA project (already mentioned above) to query this flattened view of MongoDB with SPARQL under an OWL2 QL ontology. We provide more details about this work in section 8.6.

3.3 Mapping Heterogeneous Data to RDF

In RDF-based data integration systems, the previous section has shown that ontology-based query answering needs a mapping to materialize the A-box, while ontology-based data access utilizes the same mapping to reformulate a SPARQL query into SQL. In both cases, the mapping describes the production of RDF triples from relational data.

Beyond the scope of relational databases, the question of producing RDF from existing data sources has given rise to a broad variety of approaches spanning source-specific tools, generic mapping languages and integration frameworks. Countless "RDFizers" fall in the first category[46]. Typically, they are standalone tools or Web APIs implementing ad-hoc translation of specific data formats to RDF (*e.g.* BibTex, EXIF, iCalendar, LDIF, Microformats, social networks metadata, etc.). In the second and third categories (mapping languages and integration frameworks), the concern is about mapping common data sources to RDF, in a mindset that favors formalism and genericity. These works notably pertain to XML and tabular formats (CSV/TSV, spreadsheets and relational databases). We present some of them in the subsequent sections.

Global-as-View vs. Mediator approaches. The presented approaches essentially fall in the Global-as-View mapping approach, even though we consider only one data source. This is particularly true when the RDF data produced complies with a pre-existing domain ontology: the mapping represents each concept of the ontology (*i.e.* the global schema) as a query/transformation over the concepts of the data source. Nonetheless, instead of using the term Global-as-View that refers specifically to the integration of distinct data sources, the literature often refers to such mapping approaches as *wrappers* or *mediators*.

Notable discrepancies exist in the different approaches, with regards to the way RDF triples are being produced and consumed. In the following, we refer to the g*raph materialization* and *query rewriting* approaches that we mentioned in section 1.1.

[46] https://www.w3.org/wiki/ConverterToRdf

3.3.1 Mapping XML Data to RDF

Several techniques have been proposed to perform transformations between XML and RDF, building upon various technologies such as XSLT, XQuery or schema transformations (XML Schema ⇔ OWL, DTD ⇔ OWL).

XSLT-based transformations. The **AstroGrid-D** project [Breitling, 2009] relies on a fairly simple XSLT style sheet to translate an XML file into an RDF/XML representation: URIs are built by concatenating the names of XML elements with parent-child relationship, and XML attributes are converted into properties, thereby coming up with an ad-hoc ontology. **XML Scissor-lift** [Fennell, 2014] provides greater flexibility. It leverages the XSLT-based Schematron XML validation language. Schematron rules typically report on the occurrence of an XML node. The reporting feature is diverted to generate RDF triples (in any RDF syntax), allowing for the reuse of existing ontologies. **GRDDL** [Connolly, 2007] is somewhat different from the two previous projects. It addresses the case where RDF data are embedded in XML documents, and relies on XSLT (although other transformations are possible) to extract them. A typical use case of GRDDL is the extraction from XHTML pages of data expressed in Microformats.

Schema transformations. In [Bedini et al., 2011], the authors devise a rich set of patterns that allow for the automatic transformation of XML Schema constructs into OWL assertions. The patterns are inspired from a broad set of XML Schema specifications taken from real-world B2B use cases. The goal is clearly to come up with an ontology faithfully representing the XML schema, and authors consider that not requiring an existing target ontology is beneficial compared to previous approaches. This assertion is however questionable as it ignores the reuse of existing domain knowledge formalizations. The **SPARQL2XQuery** framework [Bikakis et al., 2013] adopts a more general position. It achieves XML-to-RDF translation in two steps. First, the framework defines a mapping between an XML Schema and an OWL ontology. This step can be automatic (creation of an ad-hoc ontology like in [Bedini et al., 2011]) or manual (alignment on existing ontologies). Then, a SPARQL-to-XQuery translation is executed based on the mapping exhibited in the first step.

XQuery-based transformations. In SPARQL2XQuery and Bedini et al's approach, it is assumed that an XML Schema specification should exist in the first place. Yet, using an XML Schema is optional when working with XML data. Therefore, as an alternative, it should be possible to map XML data to RDF in the absence of any schema. This is the point of the **XSPARQL** query language [Bischof et al., 2012]. It combines XQuery and SPARQL for bidirectional transformations between XML and RDF by merging SPARQL snippets into XQuery FLWOR expressions[47]. An extension of XSPARQL provides the ability to embed SQL snippets, such that relational data can be queried and transformed into XML or RDF [Lopes et al., 2011]. Another extension supports the querying of JSON documents within XQuery [Dell'Aglio et al., 2014]. Thus, although it was initially dedicated to XML-RDF transformation, XSPARQL can now translate relational, XML, RDF and JSON data to XML or RDF. Yet, XML remains the pivot format, and the XSPARQL language itself remains rooted into XML technologies. This can seem unnatural when it comes to other formats. Furthermore, the imperative programming paradigm underlies XQuery FLWOR expressions. Utilizing this paradigm to describe the translation from XML, JSON or relational

[47] FLWOR is an imperative-style construct of XQuery, the acronym stands for "For, Let, Where, Order by, Return".

data to RDF may not be natural to some communities, *e.g.* database engineers, for whom a declarative paradigm would seem more appropriate.

3.3.2 Mapping JSON Data to RDF

Several approaches deal with the translation of JSON along with other data formats. This is the case of TARQL and RML that we describe later in section 3.3.4.

JSON-LD [Sporny et al., 2014] is designed as a syntax for the serialization of Linked Data in the JSON data format. More generally, it can be seen as a syntax to serialize directed graphs. Taken the other way round, a JSON-LD profile can be considered as a lightweight method to interpret existing JSON data as an RDF graph. Essentially, a JSON-LD context (i) associates an IRI with a JSON object, (ii) defines a mapping of JSON keys to IRIs, and (iii) allows the annotation of literal values with data types or language tags.

3.3.3 Mapping CSV, TSV and Spreadsheets to RDF

XLWrap [Langegger & Wöss, 2009] addresses the translation of tabular data files into RDF. It can cross multiple files at once, including CSV, TSV and spreadsheets from widespread software such as Microsoft Excel and OpenOffice Calc. XLWrap's mapping language is written in the RDF TriG syntax [Bizer & Cyganiak, 2014] and leverages (i) template graphs to define the shape of RDF triples, and (ii) a row-based and column-based iteration mechanism that allows to adapt to any spreadsheet layout. Indeed, spreadsheets typically contain human-readable information such as comments, titles or remarks that are irrelevant for an RDF representation, and this iteration model helps skip them. XLWrap also implements a SPARQL access to CSV, TSV and spreadsheets based on the template graphs.

The Open Refine[48] project (formerly Google Refine) helps make sense of tabular data through a set of tools to clean, transform and cross several data sets together. **RDF Refine**[49] is an extension of Open Refine meant to align the tabular data loaded in Open Refine with existing ontologies, while reconciliating the obtained URIs with third party RDF data sets [Maali et al., 2011]. In this context, RDF Refine provides an export feature that translates tabular data into RDF. The data management part of Open Refine as well as the alignment towards RDF in RDF Refine are fully performed through a Web user interface. Unfortunately, RDF Refine does not explicit nor give access to the CSV-to-RDF mapping mechanism that is enacted internally.

The **Linked CSV**[50] format is a proposition to embed in a CSV file different metadata that make it easy to link and reuse on the Web. Each row and column can be given a URI, thus providing a self-contained description of how to translate the CSV data into RDF or JSON. Other metadata can be added, for instance to specify data types, language, license, links to other related files.

[48] Open Refine: http://openrefine.org/
[49] RDF Refine: http://refine.deri.ie/
[50] Linked CSV: http://jenit.github.io/linked-csv/

CSVW is a W3C recommendation that results from the work of the *CSV on the Web* W3C Working Group[51] that has now become the *CSV on the Web* W3C Community Group[52]. Its primary goal is to abstract away from the varying syntaxes used to exchange CSV/tabular data, and to fill the lack of CSV schema description. This is achieved with a set of descriptive metadata [Pollock et al., 2015] that annotate tabular data and data collections. Contrary to the Linked CSV proposition, metadata are stored in a separate companion file, hence the existing file does not need to be modified. Besides, metadata are much richer than Linked CSV. These include alternative column names, column descriptions, mandatory/optional columns, primary keys, keywords, publisher information, etc. The recommendation comes with two additional documents describing how to generate RDF and JSON from such annotated tabular data [Tandy et al., 2015; Tandy & Herman, 2015].

3.3.4 Multi-Format Mapping Tools and RDF Integration Frameworks

Several tools and approaches were designed either as Swiss-army knifes supporting multiple data formats, or as frameworks for the integration of data sources with heterogeneous data formats.

TARQL (Transformation SPARQL) [Cyganiak, 2013] reuses SPARQL constructs to achieve the translation of RDF, CSV/TSV and JSON data formats to RDF. It performs a direct mapping: a JSON key-value pair is translated into an RDF triple by turning the key into an ad-hoc property URI, a JSON array is translated into an RDF list, but no class name is ever created. Per se, TARQL is not a mapping language, rather an experimentation that "abuses SPARQL notations to make SPARQL work for tasks it was not originally designed for", as Cyganiak puts it. He suggests that a better approach would be "to extend SPARQL with new keywords and constructs". This is precisely the approach of SPARQL-Generate that we further describe later in this section.

Datalift [Scharffe et al., 2012] provides an integrated set of tools for the publication in RDF of raw structured data (RDB, CSV, XML) and the interlinking of the resulting data sets. The process starts with a direct mapping creating ad-hoc class and predicate URIs. Then, different modules perform a-posteriori alignments with domain ontologies: RDF-to-RDF transformation by means of SPARQL queries, renaming of URIs using regular expressions, conversion of strings into URIs, automatic discovery of links with other RDF datasets with SILK[53]. Datalift does not come with a uniform mapping language but relies on the languages of the modules it builds upon. The mapping description is done interactively through a Web-based GUI, and the resulting graph is materialized and loaded into a triple store. Similarly, the **Apache Any23** project[54] provides a collection of libraries to extract structured data and metadata in RDF format from various types of documents. It provides extractors for several RDF syntaxes, HTML metadata, RDFa, Microformats, as well as CSV and XML.

DataLift and Apache Any23 do not exhibit an explicit mapping language, which hampers their reusability. As a result, extending them to support new types of data sources requires specific developments. **RML** [Dimou et al., 2014a], on the other hand, is an extension of R2RML that tackles

[51] CSV on the Web Working Group: http://www.w3.org/2013/csvw/wiki
[52] CSV on the Web Community Group: https://www.w3.org/community/csvw/
[53] SILK: http://silk.semwebcentral.org/
[54] Apache Any23: https://any23.apache.org

the mapping of data sources with heterogeneous data formats such as RDB, CSV/TSV, XML or JSON. RML and DataLift share the same concern of providing a way to interlink separate data sources at mapping time. Most approaches create links between data sets after they were translated to RDF, requiring entity-matching techniques to reconcile equivalent resources. To clear this hurdle, RML performs the mappings of different data sources simultaneously by declaratively defining joins between those sources.

SPARQL-Generate [Lefrançois & Zimmermann, 2016] has similar concerns but takes a different approach: it extends the SPARQL query language to allow for the simultaneous querying of an RDF graph and documents in non-RDF formats. Like in a SPARQL CONSTRUCT query, a SPARQL-Generate query defines the shape of RDF triples to be produced. The approach is very similar to that of XLWrap (section 3.3.3), with the advantage that SPARQL-Generate extends an existing language instead of defining a new one. Authors argue that, compared to mapping languages such as R2RML and RML, SPARQL-Generate is easier to adopt by semantic Web experts who already know SPARQL. This remark should be tempered. Firstly, RML and R2RML are written in RDF that semantic Web experts obviously know. Secondly, it is unclear whether people in charge of writing mappings for a specific data source are semantic Web specialists. In some cases, mapping designers would more likely be domain experts or database engineers who hardly know semantic Web technologies. In this context, SPARQL is probably more difficult to apprehend than declarative mapping languages, may they be written in RDF. Besides, authors contend that SPARQL-Generate allows for more concise and understandable mappings. Indeed, the goal of a graph pattern can be caught at a glance, whereas figuring out the triples generated by a set of R2RML mappings requires more attention. Nevertheless, examples provided by the authors show that the additional syntactic sugar required in SPARQL-Generate can make queries rather cumbersome. Hence, we believe that this argument should be looked at in context.

3.3.5 Mapping Relational Databases to RDF

The translation of relational data to RDF, often referred to as RDB-to-RDF, has been an active field of research during the last 10 to 15 years, as attested by several state-of-the-art surveys [Hert et al., 2011; Sequeda et al., 2011; Spanos et al., 2012; Michel et al., 2014]. Despite the variety of approaches, two major mapping methods are always considered. On the one hand, the direct mapping introspects the database schema and automatically creates an ontology that reflects its structure. On the other hand, the domain semantics-driven mapping, or custom mapping, aligns the relational schema with existing domain ontologies. In this section, we describe both approaches and review several of the works proposed in each category.

3.3.5.1 Direct Mapping

The direct mapping involves the automatic creation of an ontology (called local or ad-hoc ontology) that reflects the structure of the relational schema. It was formalized in a W3C recommendation [Arenas et al., 2012], and Sequeda et al. proposed a review of existing strategies [Sequeda et al., 2011]. Informally, the direct mapping follows simple translation rules:

- Table to Class: a table is translated into an ontological class identified by a URI “baseURI/table”;
- Column to Property: each column of a table is translated into a predicate whose URI follows the pattern “baseURI/table/column”;
- Row to Resource: each row of a table is translated into a resource whose type is the class represented by the table. Its URI is formed by using the table's primary key: “baseURI/table/primaryKey” or “baseIRI/table#primaryKey”;
- Cell to Literal: each cell with a literal value is translated into the object of a data type property;
- Cell to Resource URI: each cell with a foreign key constraint is translated into a URI that is the value of an object property.

The direct mapping may apply in different situations: when no ontology suitably describes the domain of the relational database, when the goal is the quick publication of the data in machine-readable format, or when the schema is too large to write the mappings manually. If semantic interoperability is required later on, ontology alignment methods can be used to align the local ontology with existing domain ontologies. This is the approach proposed by **DataLift**, already mentioned in section 3.3.4. **Ultrawrap** is another well-known direct mapping implementation [Sequeda & Miranker, 2013]. It defines SQL views that lay out relational data as RDF triples. Consequently, a SPARQL query can be simply rewritten into an SQL query on the SQL views.

Some approaches attempt to improve the quality of direct mapping by detecting common database design patterns. For instance, many-to-many relations are suggested by tables whose columns are all foreign keys; nullable/not nullable columns can be converted into OWL cardinality constraints; implicit subclass relationships are often suggested by a primary key used as a foreign key [Cullot et al., 2007]. Additional ontology learning techniques attempt to refine classes by exploring the data. For instance, redundant values in a column may suggest categorization patterns [Cerbah, 2008].

3.3.5.2 Domain Semantics-Driven Mapping

Commonly, relational schemas are altered by performance optimizations and pervaded with contingent maintenance history. As a result, it is difficult for a direct mapping approach to capture domain semantics that, by the way, is often encoded in the application exploiting the database. To overcome these limitations, the domain semantics-driven mapping, or custom mapping, aligns a legacy database schema with existing domain ontologies describing the same domain of interest (possibly partially). In this second category, many approaches have proposed mapping languages with varying goals and capabilities. D2RQ and R_2O were among the first projects tackling the RDB-to-RDF problem.

The **D2R** server [Bizer & Seaborne, 2004; Bizer & Cyganiak, 2006] implements the direct mapping and custom mapping approaches, and supports the rewriting of SPARQL to SQL. It builds on the **D2RQ** mapping language [Cyganiak et al., 2012] that is expressed in RDF and embeds SQL fragments to express SELECT statements or to use aggregate functions.

R_2O is a declarative XML-based language [Barrasa et al., 2004]. It addresses situations where the similarity between the relational database and the ontology is low, *e.g.* the translation of one table to several separate classes, or the join of several tables to yield resources of a single class. It provides data transformation primitives such as string manipulations, arithmetic calculations, definition of order

relations and restriction on range of values, that are rarely addressed by other languages. The **ODEMapster** query engine[55] uses an R_2O mapping either to execute the transformation in response to a query expressed in the ODEMQL query language, or to materialize the RDF graph (called massive upgrade).

Let us finally mention **Triplify** [Auer et al., 2009], an original approach that enables popular Web applications (like CMS or blog applications) to publish their relational database as Linked Data or JSON. The core idea is to map HTTP URIs to queries over the relational database. Mappings are implemented as SQL statements embedded in PHP scripts. Triplify does not support SPARQL but it can dereference URIs and return an RDF description at run-time. The RDF graph materialization is also supported.

R2RML-based approaches. In 2012, the W3C issued the **R2RML** recommendation, meant to standardize the description of customized mappings of a relational database to RDF [Das et al., 2012]. R2RML, which we describe in details in section 4.4.1, results from preliminary works held by the W3C RDB2RDF Incubator Group[56]. Having surveyed state-of-the-art RDB-to-RDF approaches such as D2RQ and R_2O [Sahoo et al., 2009], the group formalized a set of characteristics that a standard mapping language should meet [Ashok, 2009]. This gave birth to the R2RML recommendation that covers the mapping language specification but does not tackle the implementation issue. Compliant tools may therefore adopt varying strategies. Today, R2RML has reached a notable consensus[57]. Below we shortly review some of the implementations.

Morph-RDB [Priyatna et al., 2014] is an implementation of R2RML by the developers of R_2O and ODEMapster. It supports the graph materialization and the rewriting of SPARQL to SQL for MySQL and PostgreSQL. Two other versions of the product rely on R2RML to deal with other tabular data sources: Morph-streams deals with streamed data typically produced by sensors [Calbimonte et al., 2012], and Morph-GFT queries Google Fusion Tables [Priyatna et al., 2013]. The authors also proposed a method to automatically generate an R2RML mapping complying with the direct mapping principles [de Medeiros et al., 2015].

As a standard, R2RML has caught much attention from projects that predated the recommendation such as the aforementioned XSPARQL and Ultrawrap projects. In **XSPARQL,** an R2RML mapping is read as a regular input RDF graph. An appropriate query document then interprets the mapping alongside a relational database to generate the corresponding RDF triples [Lopes et al., 2011]. Capsenta's Web site reports that **Ultrawrap** supports R2RML[58]. **R2RML-F** proposes to incorporate domain logic in R2RML mappings by means of custom ECMAScript functions [Debruyne & O'Sullivan, 2016]. Some works have investigated the usability and appropriateness of R2RML for users with varying backgrounds. In this regard, the need for graphical editors assisting users in the design of R2RML mappings has been pointed out [Pinkel et al., 2014; Sengupta et al., 2013]. Alternatively, Stadler et al. have proposed the **Sparqlification Meta Language** (SML), equal in expressiveness to R2RML but with

[55] http://neon-toolkit.org/wiki/ODEMapster
[56] W3C RDB2RDF Incubator Group: http://www.w3.org/2005/Incubator/rdb2rdf/
[57] R2RML Implementations: http://www.w3.org/2001/sw/rdb2rdf/wiki/Implementations
[58] https://capsenta.com/wp-content/uploads/2014/12/UltrawrapDatasheet_Final.pdf

a more compact syntax [Stadler et al., 2015]. The study conducted by the authors tends to show that SML is more human-friendly and easily adopted by non-R2RML-expert users.

Beyond the academic community, it is interesting to notice that well-known commercial solutions have adopted R2RML as a full-fledged building block within large software suites. **Oracle RDF Semantic Graph** is an option of Oracle Database Enterprise Edition that focuses on the storage, querying of and reasoning on relational data alongside RDF graphs. In this context, R2RML and the direct mapping are supported to define RDF views over relational data. **Virtuoso Universal Server**[59] is a commercial and open-source tool suite developed by OpenLink. It implements RDB-to-RDF transformations by mapping relational data to so-called Linked Data Views, using the proprietary Meta Schema Language (MSL). The support of R2RML is achieved by compiling an R2RML mapping into MSL.

3.4 Conclusion

The goal we pursue in this thesis is to enable RDF-based data integration over heterogeneous data sources. In this chapter, we first reminded the main principles of data integration, and we made a focus on ontology-based data integration approaches. Then, we carried out a review of previous works dedicated to the translation of existing data formats and databases into RDF. In all these approaches, the fundamental keystone is the description of mappings between a data source and a global schema.

To enable the mapping to RDF of data sources with heterogeneous data models and query capabilities, we need a flexible mapping description method. In this respect, the above reviewed approaches provide valuable groundwork, but they are generally suited to a specific type of data source or a specific data integration task. Moreover, while some of them define an explicit mapping language, others keep the mapping details buried in application code. In the next chapter, we analyze in further details the requirements that a generalized mapping language should meet in order to transparently enable the mapping of data items from heterogeneous data sources into an arbitrary RDF representation.

Note that this chapter focused on data integration and translation to RDF. Our goal is nonetheless to propose a method to rewrite SPARQL queries into varying target query languages, and to demonstrate the approach in the case of the MongoDB document store. Therefore, we complement this chapter by reviewing in further details state-of-the-art SPARQL rewriting methods in section 6.2, and other works related to MongoDB in section 7.1.

[59] Virtuoso Universal Server: http://www.w3.org/wiki/VirtuosoUniversalServer

Chapter 4. Underpinning a Generalized Mapping Language

4.1 Introduction

Different kinds of databases typically differ in several aspects: the data model that underlies the data structures, the access method and query language used to retrieve data, the cross-referencing scheme, the data exchange formats, etc. Although it seems illusory to seek universal support of any database, our undertaking is to define an approach that would enable the mapping of a large and extensible scope of relational and non-relational databases to RDF, in a flexible and transparent manner.

In this chapter, we investigate what "good properties" a generalized mapping language should have to fulfill this goal, and we formalize this result as a set of six requirements (section 4.2). Then, we evaluate some mapping languages proposed in previous works against those requirements, and we justify why we choose R2RML, complemented with RML, R2RML-F and CSVW, as the underpinning of a generalized mapping language (section 4.3). Finally, we review the main capabilities of R2RML and RML (section 4.4), as an introduction to the next chapter that defines our proposition for a generalized mapping language, namely xR2RML.

4.2 Requirements for a Generalized Mapping Language

4.2.1 Data models

The landscape of modern database systems evidences a vast diversity of data models. In order to describe the translation of various data models to RDF, a mapping language must be able to reference any data element within those data models. This ability depends on the data model itself, but more importantly on the data exchange formats available through the database APIs. Below, we explore common data models as well as the data exchange formats exposed by common database APIs, and we figure out how a mapping language can reference data elements within data retrieved through those APIs.

- **Row-Column-based systems**: Relational databases comply with a row-based model in which column names uniquely reference cells in a row. NoSQL extensible column stores[60] also comply with the row-based model, with the difference that all rows do not necessarily share the same columns (e.g. BigTables, Cassandra). For such systems, referencing data elements is simply achieved using column names.
- In databases relying on a specific **data representation format** like **JSON** (notably in NoSQL document stores such as MongoDB and CouchDB) or **XML** (like in native XML databases BaseX and

[60] aka. column family store, column-oriented store, etc.

eXistDB), data is stored and retrieved using that specific data format. Insofar as such data formats can hardly be reduced to a row-based model, there generally exist appropriate languages to parse and reference data elements within documents. Typically, JSONPath will be used to parse JSON documents, and XPath to parse XML documents.

- **Object-oriented** databases conventionally provide methods to serialize objects, typically as key-value associations: keys are attribute names while values are objects (representing composition or aggregation relationships), or compound values (collection, map, etc.). Serialization is typically done in common data exchange formats such as XML or JSON. Thereby, the problem of accessing data elements of object values is reduced to the simpler problem of accessing data elements within a common data format. Here again, XPath or JSONPath will apply to an object serialized in XML or JSON.
- A **directory data model** is organized as a tree in which each node has an identifier and a set of attributes represented as "`name=value`". Attributes hold literal values and cannot be compound. A search query retrieves a set of matching nodes along with their attributes. For instance, each entry retrieved from an LDAP search request is named after its LDAP path, *e.g.* "`cn=Franck Michel,ou=cnrs,o=fr`", followed by its attributes in the form "attribute=value". Referencing data elements within each entry is simply achieved using attribute names. This flat structure very much resembles a tabular view in which values are accessed using column names.
- In **graph databases**, the abstract data model consists of nodes connected by edges. Two types of query are generally provided, that return either a set of values extracted from a graph, or a set of nodes and edges (a sub-graph or any constructed result graph). We illustrate those two options:

  (i) The *SPARQL Query Results* standard proposes methods to encode, in XML, JSON and CSV/TSV, the binding of variables and values in query results of SELECT and ASK queries. In the same idea, the Neo4J graph database proposes a JDBC interface to return a SPARQL SELECT result set as a regular relational table.

  (ii) Alternatively, a query may return a set of nodes and edges representing a result graph. For instance, a SPARQL CONSTRUCT query returns an RDF graph constructed by substituting variables in a set of triple templates. However, again, databases generally provide APIs to serialize a result graph. To stick to the example of Neo4J, its REST interface returns result graphs serialized as JSON documents.

  Hence, although a graph may be a somehow complex data structure, and whether a query returns a set of values or a graph, query results can be fairly easy to manipulate using well-known formats: a row-column model, a serialization in JSON or some other data exchange format.

It finally occurs that the way a mapping language can reference data elements within query results depends on the API capabilities as much as the data model itself. Hence, we come up with a first requirement for a generalized mapping language:

> ***Requirement 1**: **Open set of data element reference syntaxes***
>
> *A generalized mapping language should transparently deal with any type of data element reference syntax. This shall typically include a column name (applicable to row-based data models as well as any row-based query result), JSONPath, XPath, etc. A mapping processing engine shall be able to evaluate such expressions against query results, but the mapping language itself must remain free from any explicit reference to specific expression syntaxes.*

Note that an alternative to this requirement is to define a pivot format: documents with heterogeneous data formats are first translated into the pivot format, then an appropriate evaluation language extracts data elements from documents in the pivot format. This way, only one syntax has to be supported by the query language. This is the choice made by XSPARQL [Dell'Aglio et al., 2014] to support JSON documents: XSPARQL function `xsparql:json-doc` first translates JSON documents into an equivalent XML representation. Then, XPath expressions in the mapping are evaluated against this XML representation. Consequently, XPath is the only data element reference syntax that is used explicitly in an XSPARQL query. This has the advantage of presenting a homogeneous mapping language, but comes at the cost of translating documents to XML first. Instead, what *Requirement 1* proposes is to keep the document in its native format, and use an appropriate syntax explicitly in the mapping language, like JSONPath in this example.

4.2.2 Query languages

Parallel to the variety of data models, we observe an even larger heterogeneity of modern databases with regards to query languages. Relational databases all support ANSI SQL, and most XML databases support XQuery and XPath. By contrast, NoSQL is a catchall term referring to profoundly diverse systems [Gajendran, 2013; Hecht & Jablonski, 2011]. They have heterogeneous access methods ranging from low-level APIs to expressive declarative query languages. Despite several propositions of common query languages (N1QL[61], UnQL[62], SQL++ [Ong et al., 2014], ArangoDB Query Language[63], CloudMdsQL [Kolev et al., 2014]), no consensus has emerged yet, that would fit most NoSQL databases.

Therefore, until a standard eventually arises, we define the following requirement for a generalized mapping language:

[61] http://www.couchbase.com/communities/n1ql
[62] http://unql.sqlite.org/index.html
[63] http://docs.arangodb.org/Aql/README.html

> ***Requirement 2**: **Open set of query languages***
>
> *A generalized mapping language should allow defining a data source as the content of a document (typically accessed by URL) or the result of executing an arbitrary query against a database; the query should be expressed in any query language that is appropriate for that database.*
>
> *A mapping processing engine shall document the access methods, databases and query languages it supports, but the mapping language definition must remain free from any explicit reference to specific query languages: a query is a mere character string, with no explicit reference to the type of language.*

This requirement relies on the assumption that the databases we may want to translate to RDF all provide a query language. When it comes to NoSQL databases, this assumption may be somewhat restrictive. For instance, NoSQL key-value stores, such as Redis[64], DynamoDB[65] or Coubase Server[66], come with simple key-based operations (typically put, get, delete) by means of APIs for imperative programming languages. If such a system is to be mapped to RDF using a mapping language, a mapping processing engine should implement a mechanism to bridge this gap, typically by defining a query syntax to be compiled into a programming language.

Similarly, most NoSQL column stores and document stores expose a Hadoop query interface, where Map-Reduce jobs must be written using APIs for common programming languages. Here again, bridging this gap requires defining a declarative query language that, in turn, can be compiled into a programming language. Interestingly enough, this is the goal of the manifold SQL-on-Hadoop projects that were born during the last years. For instance Apache Hive[67], Apache Drill[68], Spark SQL for Apache Spark[69] and Cloudera Impala[70] provide a subset of SQL as a query language on top of Hadoop implementations, by compiling SQL queries into Map-Reduce jobs passed on to the underlying infrastructure [Floratou et al., 2014].

Furthermore, beyond databases equipped with a query language, many data sources are also available on the Web, that do not provide a query language. These are typically formatted files that one can download or retrieve by invoking a Web service. To deal with those data sources, it must be possible to fetch the whole data at once. Hence the first part of the definition: “defining a data source as the content of a document (typically accessed by URL)”. To allow this, a generalized mapping language should simply enable setting an empty query string or no query at all.

4.2.3 Collections

Many data models support the representation of collections: these can be sets, arrays or maps of all kinds (sorted or not sorted, with or without duplicates, etc.). Although the RDF data model supports

[64] Redis: http://redis.io/
[65] DynamoDB: https://aws.amazon.com/fr/dynamodb/
[66] Couchbase Server, formerly known as Membase: http://www.couchbase.com/
[67] Apache Hive: https://hive.apache.org/
[68] Apache Drill : http://drill.apache.org/
[69] Spark SQL: https://spark.apache.org/sql/
[70] Cloudera Impala: http://www.cloudera.com/products/apache-hadoop/impala.html

such data structures, many existing mapping languages do not allow for the production of RDF collections (`rdf:List`) and containers (`rdf:Bag`, `rdf:Seq`, `rdf:Alt`). Instead, structured values such as collections or key-value associations are flattened into multiple RDF triples. We illustrates this concern in the example below that depicts an XML collection consisting of two "movie" elements:

```
<director name="Woody Allen">
    <movie>Annie Hall</movie>
    <movie>Manhattan</movie>
</director>
```

Figure 6 illustrates two possible translations of this collection. On the left, each "movie" element entails a separate triple, whereas on the right, both "movie" elements are turned into an RDF container of type Sequence.

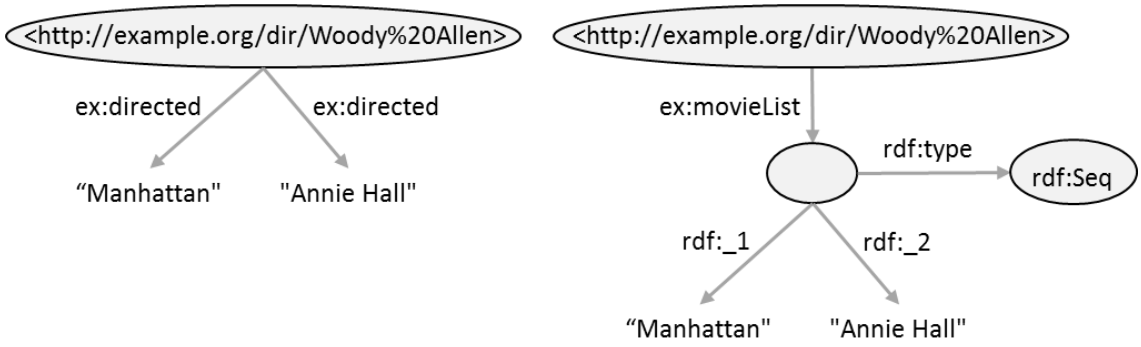


Figure 6: Translation of a collection into multiple RDF triples (left), to a container (right)

Furthermore, some data models use cross-references to model one-to-many relationships (discussed in the next section). Some existing mapping languages can follow cross-references to produce multiple triples from a one-to-many relationship, but again, they cannot turn them into RDF collections or containers.

These observations lead us to a third requirement for a generalized mapping language:

> ***Requirement 3: Generate RDF collections and containers***
>
> *In order to map heterogeneous data to RDF while preserving the semantics of collections and one-to-many relationships, a generalized mapping language should be able to map them to RDF collections (list) and containers (bag, seq, alt).*

4.2.4 Cross-references

Cross-references allow modelling relationships between different logical entities of a database, or across several databases. They are implemented differently in different contexts: in relational data model, a cross-reference is materialized by a column holding values of another column (usually a primary key); in a NoSQL document store, a field within a JSON document may refer to another document using *e.g.* its identifier or any other field; within a graph-based database, an arc is the materialization of a cross-reference between several nodes.

A database may provide technical means to ensure the integrity of cross-references, such as foreign key constraints in relational data models, and to ensure the efficient processing of queries involving cross-references by setting up indexes.

The semantics beard by a cross-reference can be implicitly encoded in a data model, like in the case of a relational schema, or in an object-oriented model where a cross-reference usually depicts object aggregation and composition relationships. In an RDF graph, this semantics is made explicit by arc labels.

When translating data to RDF triples, a cross-referenced logical entity may be mapped alternatively to the subject or the object of RDF triples. We illustrate this case in Figure 7 that depicts two relational tables with a foreign key constraint from column `Resource_id` of table `Study` to the primary key of table `Resource`.

Figure 7: Relational schema with a cross-reference

A basic translation to RDF triples may relate the URI for study 10 with the URI for resource 4. For instance:

```
<http://example.org/study#10> ex:involves <http://example.org/resource#4>.
```

This translation mimics the relational model. Now, let us assume our target ontology defines `ex:involves` as a property whose object is not the resource URI, but the resource name. We may now wish to generate the following triple:

```
<http://example.org/study#10> ex:involves "Res1".
```

Intuitively, we notice that generating the expected triple requires running a join query between the two tables. More generally, this sheer example emphasizes that a generalized mapping language should enable the description of a join query between database logical entities (rows, documents…) in order to produce the appropriate RDF triples. Hence the requirement below:

> ***Requirement 4**: **Follow cross-references***
>
> *A generalized mapping language should allow the description of join queries to retrieve cross-referenced logical entities.*

4.2.5 Custom functions

Values stored in a database may be encoded according to application-dependent patterns. Retrieving relevant data in order to produce the appropriate RDF terms may require parsing database values with regular expressions or any other custom function to make domain logic explicit.

Furthermore, in real world applications, it is common to store values using a format that the database cannot natively manage. For instance, in key-value stores and in extensible column stores, values are

stored as binary objects whose content is opaque to the system; in a relational database, an application designer may choose to embed JSON or CSV values for performance concerns or to facilitate data exchange with other components of the application.

Although a database query language may be used to compute appropriate values, for instance using SQL string manipulation functions, a generalized mapping language should make it possible to apply custom functions to values retrieved from the database.

Hence the definition of requirement 5:

> ***Requirement 5*: *Custom functions***
>
> *A generalized mapping language should enable the use of custom functions implementing domain logic in order to compute appropriate RDF terms. In particular, it must be able to deal with the mixing of different data formats, such as an embedded XML value within a relational column.*

4.2.6 Organizing RDF datasets with named graphs

The abundance of data sources that we may potentially want to translate to RDF questions the organization of the generated RDF datasets. Various solutions may be envisaged to address this concern; one of them, proposed in the SPARQL 1.0 recommendation [Harris & Seaborne, 2008] and slightly extended in RDF 1.1 [Cyganiak et al., 2014], consists in defining an RDF dataset as a set of graphs containing at least one default graph, and zero or more named graphs identified by an IRI. This leads us to a last requirement:

> ***Requirement 6: Named Graphs***
>
> *A generalized mapping language should allow assigning generated RDF triples to a named graph identified by its IRI, or to the default graph if no graph IRI is provided.*

4.3 Underpin a Generalized Mapping Language with State-of-the-Art Works

In the previous section, we have exhibited a set of requirements that a generalized mapping language should meet in order to support the mapping of heterogeneous databases to RDF. In Table 2, we compare mapping languages proposed in state-of-the-art works (presented in Chapter 3) with respect to these requirements. This shall help us figure out which of them have good properties that we can rely on to define a generalized mapping language.

Table 2 is sorted as follows: mapping languages dedicated to RDBs are listed first (R_2O, D2RQ, SML, R2RML, R2RML-F); then comes RML that supports RDBs as well as several other data formats; lastly we list mapping languages dedicated to XML data sources (XSPARQL and SPARQL2XQuery), and those dedicated to CSV (CSVW and XLWrap).

Table 2: Comparison of mapping languages proposed in state-of-the-art works against the requirements for a generalized mapping language

Requirement / Language	1: Open set of data element reference syntaxes	2: Open set of query languages	3: Generate RDF collections and containers	4: Follow cross-references	5: Custom functions	6: Named graphs	No. of reqmts. met
R_2O [Barrasa et al., 2004]	✗ (Column)	✗ (SQL)	✗	✓	✓ (proprietary)	✓	3
D2RQ [Bizer & Cyganiak, 2006]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
SML [Stadler et al., 2015]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
R2RML [Das et al., 2012]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
R2RML-F [Debruyne & O'Sullivan, 2016]	✗ (Column)	✗ (SQL)	✗	✓	✓ (ECMAscript)	✓	3
RML [Dimou et al., 2014a]	✓	✗	✗	✓	✗	✓	3
XSPARQL [Bischof et al., 2012]	✗ (XPath)	✗ (XQuery, SPARQL)	✓ (XQuery)	✓ (XQuery)	✓ (XPath)	✗	3
SPARQL2XQuery [Bikakis et al., 2013]	✗ (XPath)	✗ (XQuery)	✗	✓ (XQuery)	✓ (XPath)	✓	3
CSVW [Tandy et al., 2015]	✗ (Column)	✗	✓ (rdf:List only)	✓	✓	✗	3
XLWrap [Langegger & Wöss, 2009]	✗ (Row/Col.)	✗	✗	✓	✓ (OpenOffice Calc)	✗	2

We now analyze each requirement in more details:

- Requirement 1 (multiple data element reference syntaxes) is only supported by RML. This is not surprising since the other mapping languages reviewed here are dedicated to a specific type of data source or data format, whereas RML applies to several data formats.
- Requirement 2 (multiple query languages) is not supported by any of the reviewed mapping languages.
- Requirement 3 (generation of RDF collections and containers) is addressed in existing mapping languages either partially or in a somewhat restricted context. TARQL [Cyganiak, 2013] performs the direct mapping of JSON documents, where each key-value pair is translated into an RDF triple with ad-hoc properties, while an array is translated into an RDF collection. CSVW [Tandy et al., 2015] allows for the generation of collections but containers are not supported. Lastly, XSPARQL [Bischof et al., 2012] describes the construction of an RDF query result using XQuery FLWOR

expressions. Thus, it is easy to iterate on a set of XML elements and produce the members of an RDF collection or container. Therefore, the feature is supported by the XQuery query language itself in an imperative manner.

- Requirement 4 (cross-references): this feature is generally well supported, although a closer look evidences significant discrepancies in the way it is implemented. In XSPARQL and SPARQL2XQuery, the support of cross-referenced entities is not materialized in the mapping language; instead, it builds on the ability of XQuery to compute join queries over documents of an XML database. Therefore, although the feature is supported, the mapping language is closely bound to the XQuery query language itself, which may not be appropriate for the definition of a generalized mapping language.
- Requirement 5 (custom functions): the manipulation of data source values is supported in different ways. R_2O provides proprietary functions, while some others rely on the database query language: mapping languages dedicated to RDBs may create a SQL query to compute an appropriate value with SQL functions (this is the case of D2RQ and R2RML), and languages dedicated to XML may do the same using XPath expressions (XPath is a subset of XQuery). XLWrap relies on functions available to deal with spreadsheets in OpenOffice Calc. Finally, R2RML-F [Debruyne & O'Sullivan, 2016] proposes to point to an ECMAScript implementing the processing, and CSVW opens up further possibilities by allowing to point to any type of script or template (with properties `url`, `scriptFormat` and `targetFormat`).
- Requirement 6 (named graphs) is not supported by XSPARQL, CSVW and XLWrap.

XSPARQL and SPARQL2XQuery both strongly rely on the capabilities and expressiveness of the underlying XQuery/XPath language, as discussed above in requirements 3 and 4. Hence, they may be appropriate candidates for a generalized mapping language, with the constraint that the mapping language would derive from and extend XQuery.

CSVW and XLWrap both define a vocabulary for the mapping of CSV documents. XLWrap proposes an approach based on template graphs, similar to what could be written using SPARQL CONSTRUCT queries. However, it is closely linked to the format of spreadsheets, using notations such as A2, C4 etc. to refer to cells or range of cells in a table, and elements such as `xl:FileRepeat`, `xl:RowShift`, `xl:ColShift` and `xl:SheetShift` that are inappropriate for a wider scope of data sources. Yet, one could argue that the language could be extended with less specific properties. CSVW, on the other hand, is a more general approach. Its primary goal is to abstract away from the varying syntaxes used to exchange CSV/tabular data, and to fill the lack of CSV schema description. This is achieved with a set of descriptive metadata [Pollock et al., 2015] used to annotate tabular data and data collections. Once CSV data is annotated, it can be processed in various ways, notably the translation into RDF. CSVW has commonalities with R2RML like the modelling of cross-references. Furthermore, its recommendation to handle custom parsing functions is elegant and flexible.

In the group of mapping languages dedicated to relational databases, R2RML stands out as a reference point: as a W3C recommendation, it is the result of a large consensus, and leverages the experience of previous projects, including D2RQ. Dating back to 2012, it is now a well-accepted standard with mature

implementations[71], and several older projects have made extensions to comply with R2RML, *e.g.* D2RQ, Virtuoso and R_2O. More importantly, there exist some extensions of interest for our undertaking: RML extends R2RML to address a wider range of data formats including CSV, XML and JSON; R2RML-F improves it with the support of external ECMAScript functions, and both are backward compatible with R2RML. Therefore, when we consider the group {R2RML, RML, R2RML-F}, altogether they cover four of our requirements. Besides, R2RML-F may be extended to support a larger set of scripts and templates, as proposed by CSVW.

Conclusion. As an outcome of the above discussion, we propose to use R2RML as the underpinning of xR2RML, a proposition for a generalized mapping language. Elements of RML can be included to cope with heterogeneous data formats, and propositions of R2RML-F and CSVW can be leveraged to support custom functions in the form of an extensible set of scripts and templates.

Arguably, these are not the only options to underpin the design of a generalized mapping language. Following the example of XSPARQL, XQuery could be selected as a starting point. An experimentation [Dell'Aglio et al., 2014] presents an extension of XSPARQL to support the querying of JSON documents, within XQuery queries. The method translates JSON documents into an equivalent XML representation, XPath expressions are then used to evaluate the XML document. This approach has the advantage of presenting a homogeneous mapping language. Yet, while it could certainly be extended towards other data formats and databases, the translation to XML is a step that can be saved if the appropriate language is used in the first place: JSONPath, in this example. Additionally, as an extension of XQuery, XSPARQL describes mappings in an imperative fashion, using XQuery FLWOR constructs. Conversely, R2RML is a declarative set of mappings, and we argue that a declarative mapping language would be better suited for the adoption by users with no computer-programming background. This assertion should however be supported by an evaluation study based, for instance, on cognitive dimensions frameworks such as described by [Blackwell & Green, 2000].

In the next section, we recall R2RML and RML capabilities.

4.4 R2RML and RML

4.4.1 R2RML: RDB-to-RDF mapping

R2RML [Das et al., 2012] is a W3C recommendation meant to describe customized mappings of a relational database into an RDF data set. An R2RML mapping is expressed as an RDF graph written in the RDF Turtle syntax [Beckett et al., 2014]. An R2RML mapping graph consists of *triples maps*, each one specifying how to map rows of a logical table to RDF triples.

A triples map is composed of exactly one *logical table* (property `rr:logicalTable`), one *subject map* (property `rr:subjectMap`) and any number of *predicate-object maps* (property `rr:predicateObjectMap`). A logical table may be a table, an SQL view (property `rr:tableName`), or the result of a valid SQL query (property `rr:sqlQuery`). A predicate-object map consists of *predicate maps*

[71] R2RML implementations: https://www.w3.org/2001/sw/rdb2rdf/wiki/Implementations

(property `rr:predicateMap`) and *object maps* (property `rr:objectMap`). For each row of the logical table, the subject map generates a subject IRI, while each predicate-object map creates one or more predicate-object pairs. Triples are produced by combining the subject IRI with each predicate-object pair. Additionally, triples are generated either in the default graph or in a named graph specified using *graph maps* (property `rr:graphMap`).

Subject, predicate, object and graph maps are all R2RML *term maps*. A term map is a function that generates RDF terms (either a literal, an IRI or a blank node) from elements of a logical table row. A term map must be exactly one of the following: a *constant-valued term map* (property `rr:constant`) always generates the same value; a *column-valued term map* (property `rr:column`) produces the value of a given column in the current row; a *template-valued term* map (property *rr:template*) builds a value from a template string that references columns of the current row. There exist shortcut notations for constant-valued term maps: `rr:subject`, `rr:predicate`, `rr:object` and `rr:graph`.

When a logical resource is cross-referenced, typically by means of a foreign key relationship, it may be used as the subject of some triples and the object of some others. In such cases, a *referencing object map* uses IRIs produced by the subject map of a (parent) triples map as the objects of triples produced by another (child) triples map. In case both triples maps do not share the same logical table, a join query must be performed. A join condition (property `rr:joinCondition`) names the columns from the parent and child triples maps, that must be joined (properties `rr:parent` and `rr:child`).

As an illustration, let us consider the relational database depicted in Figure 8. It consists of two tables, representing movies and the directors. A foreign key constraint has been defined for column `MOVIE_NAME` in table `DIRECTOR`, whose values must be that of column `NAME` in table `MOVIES`.

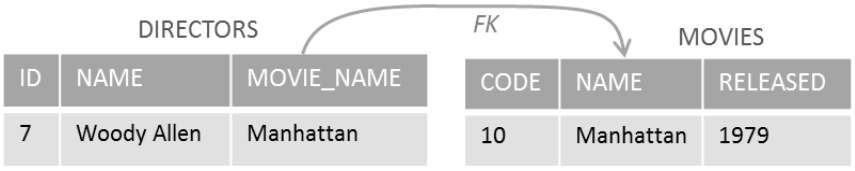


Figure 8: Example of the Movies and Directors relational schema

We now consider the R2RML mapping graph in Listing 1. Triples map `<#R2RML_Movies>` uses table `MOVIES` to create triples linking each movie (whose URI is built with the movie code) with their name (property `dc:title`) and release date (property `dc:date`). Triples map `<#R2RML_Directors>` uses table `DIRECTORS` to create triples linking each movie director (whose IRIs is built with column `NAME`) with the movies he/she directed. This involves a reference object map and a join condition between columns `DIRECTORS.MOVIE_NAME` and `MOVIES.NAME`.

```
<#R2RML_Movies>
  rr:logicalTable [ rr:tableName "MOVIES"; ];
  rr:subjectMap [ rr:template "http://example.org/movie/{CODE}"; ];
  rr:predicateObjectMap [
     rr:predicate dc:title;
     rr:objectMap [ rr:column "NAME"]
  ];
  rr:predicateObjectMap [
     rr:predicate dc:date;
     rr:objectMap [rr:column "RELEASED"; rr:datatype xsd:date ]
  ].

<#R2RML_Directors>
  rr:logicalTable [ rr:tableName "DIRECTORS"; ];
  rr:subjectMap [ rr:template "http://example.org/dir/{NAME}"; ];
  rr:predicateObjectMap [
     rr:predicate ex:directed;
     rr:objectMap [
        rr:parentTriplesMap <#R2RML_Movies>;
        rr:join [
           rr:child "MOVIE_NAME";
           rr:parent "NAME";
        ]
     ]
  ].
```

Listing 1: Example of R2RML triples map

4.4.2 The RML extension of R2RML

RML [Dimou et al., 2014b, 2014a] is an extension of R2RML that targets the mapping of data sources with heterogeneous data formats such as CSV/TSV, XML or JSON.

An RML logical source (property `rml:logicalSource`) extends the R2RML logical table and points to the data source (property `rml:source`): this may be a file on the local file system, or data returned from a Web service for instance. Naming the data source within the mapping makes it possible to simultaneously map several related data sources. Indeed, different data sources may use different identifiers to refer to the same logical entity. If those sources are mapped independently of each other, a subsequent step may be necessary to identify and link the different URIs referring to the same logical entities (named entity reconciliation). Conversely, RML links data sets at mapping time by enabling the simultaneous mapping of multiple data sources, thereby allowing for cross-references between logical entities defined in various data sources.

A *reference formulation* (property `rml:referenceFormulation`) names the syntax used to reference data elements within the logical source. As of today, possible values are `ql:JSONPath`, `ql:XPath`, `ql:CSS3` and `rr:SQL2008`. Data elements are referenced with property `rml:reference` that extends R2RML property `rr:column`. Its object is no longer just a column name but can be an expression whose syntax matches the reference formulation. Similarly, the definition of R2RML property `rr:template` is extended to allow for such reference expressions to be enclosed within curly braces ('{' and '}').

Documents addressed by RML can be relational tables as well as documents consisting of compound values, such as JSON or XML tree values. The default iteration model of R2RML is row-based. In the case of compound values, this model may no longer be relevant: for instance, it may be necessary to iterate on the top-level elements of an XML document instead of just considering the XML document as a whole. To address this issue, RML introduces the `rml:iterator` whose object is an expression matching the reference formulation.

We illustrate RML with an example very similar to the R2RML snippet above. We consider the two JSON files listed below:

movies.json:

```
[  {"name": "Manhattan", "code": "Manh"},
   {"name": "In the Mood for Love", "code": "Mood"}
]
```

directors.json:

```
[  {"name": "Woody Allen", "directed": ["Manhattan", "Annie Hall"]},
   {"name": "Wong Kar-wai", "directed": ["2046", "In the Mood for Love"]}
]
```

Listing 2 depicts a possible RML mapping graph for those files: in the two triples maps `<#RML_Movies>` and `<#RML_Directors>`, the reference formulation is set to JSONPath. Each file consists of a JSON array. To deal with each movie and director individually, the `rml:iterator` property changes the default iteration pattern with the JSONPath expression "$.*", that iterates on each element of the arrays.

4.4.3 Discussion

RML improves R2RML with the ability to deal with various data formats. To do so, it generalizes the reference to a database value from a column name to an expression using the appropriate language, such as XPath and JSONPath. Nevertheless, RML does not investigate the constraints that arise when dealing with different types of databases, each one having its own query language. In some cases, the language used to query the database may be the same as the language used to extract data elements from query results. For instance, XPath can be used to query an XML native database, as well as to extract data elements from XML query results. In the general case however, both languages must be dissociated, e.g. the MongoDB NoSQL document store uses a proprietary JavaScript-based query language, while results are JSON documents that shall be evaluated against JSONPath expressions.

Furthermore, RML explicitly refers to supported evaluation languages (`ql:JSONPath, ql:XPath`). Thus, supporting a new evaluation language requires changing the mapping language definition. To achieve more flexibility, we believe that the mapping language should remain free from any explicit reference to a query language. Instead, such characteristics should be implementation-dependent: typically, each implementation should provide guidelines with respect to what query language(s) it supports for what data source.

```
<#RML_Movies>
  rml:logicalSource [
    rml:source "movies.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{*.code}" ];
  rr:predicateObjectMap [
     rr:predicate dc:title;
     rr:objectMap [ rr:column "$.name"]
  ];
  rr:predicateObjectMap [
     rr:predicate dc:date;
     rr:objectMap [rr:column "$.released"; rr:datatype xsd:date ]
  ].

<#RML_Directors>
  rml:logicalSource [
     rml:source "directors.json";
     rml:referenceFormulation ql:JSONPath;
     rml:iterator "$.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/dir/{$.name}"; ];
  rr:predicateObjectMap [
      rr:predicate ex:directed;
      rr:objectMap [
         rr:parentTriplesMap <#R2RML_Movies>;
         rr:join [
            rr:child "$.directed.*";
            rr:parent "$.name";
         ]
      ]
  ].
```

Listing 2: Example of RML triples map

4.5 Conclusion

In this chapter, we analyzed what properties a generalized mapping language should have in order to enable the mapping of a large and extensible scope of databases to RDF. We formalized this result as the following set of requirements:

1. Allow an open set of data element reference syntaxes, including at least a column name, JSONPath and XPath. The mapping language vocabulary must remain free from any explicit reference to specific syntaxes.
2. Allow an open set of database query languages. The mapping language must remain free from any explicit reference to specific query languages. Typically, supporting a new database coming with its own query language merely requires the implementation of a database connector, but no changes are needed in the definition of the mapping language itself.
3. Generate RDF collections (list) and containers (bag, seq, alt) to preserve the semantics of collections and one-to-many relationships.

4. Allow the description of join queries to retrieve cross-referenced logical entities.
5. Enable the use of custom functions implementing domain logic to compute appropriate RDF terms.
6. Allow assigning generated RDF triples to a named graph or to the default graph.

Taken the other way round, data sources to be mapped to RDF using such a generalized mapping language need to fulfil some requirements:

- The database interface should provide a declarative query language. If not, it must be possible to fetch the whole data at once, such as a file being downloaded or retrieved from a Web service.
- There must exist technical means to parse query results in order to extract data elements from them. Such means may range from sheer column names to more expressive languages like XPath or JSONPath.

Later on, we evaluated some mapping languages proposed in previous works against those requirements, and we showed that R2RML, complemented with some elements from RML, R2RML-F and CSVW, is a good starting point to underpin a generalized mapping language. Finally, we reviewed the main capabilities of R2RML and RML.

In the next chapter, we shall propose the definition of an actual generalized mapping language, called xR2RML.

Chapter 5. The xR2RML Mapping Language

5.1 Introduction

The Web of Data results of the publication and interlinking of various open data sets on the Web, motivated by the idea that making heterogeneous data available in a common machine-readable format should create opportunities for novel applications and services. To a great extent, the success of this undertaking depends on the ability to reach out data locked in hidden data silos (conventionally called the deep Web [He et al., 2007]) that keep growing ever faster as data is continuously accumulated in more and more heterogeneous databases. In particular, the overwhelming success and popularity gained by NoSQL systems during recent years should urge the Web of Data community to regard NoSQL systems as valuable potential contributors of linked open data.

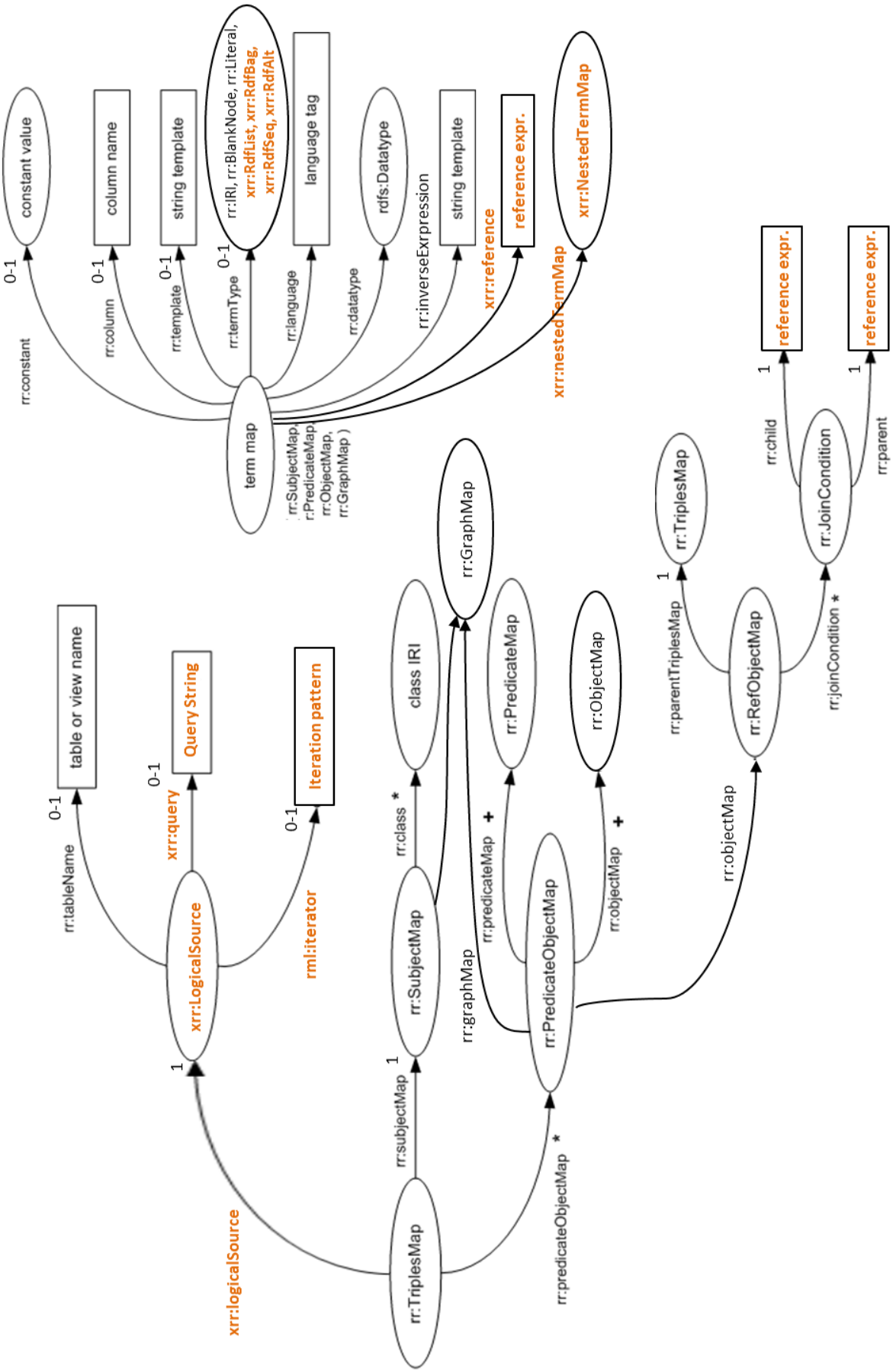
In Chapter 4, we proposed a set of requirements that a generalized mapping language should comply with in order to enable the mapping of a large and extensible scope of databases to RDF. In this chapter, we present such a generalized mapping language, namely xR2RML[72]. Following the analysis we conducted in section 4.3, several previous works underpin the design of xR2RML. First, it extends R2RML, a standard, well-adopted mapping language for relational databases. Second, to address the mapping of heterogeneous data formats, we leverage some propositions of RML that is itself an extension of R2RML. Third, we identified R2RML-F and CSVW as good candidates to implement domain logic by means of custom functions. In this chapter, we address the question of custom functions in a proprietary manner. Yet, in section 5.8, we propose a modelling wherein xR2RML could benefit from those works.

This chapter illustrates the features of xR2RML along with the reasons for the choices we made. A prototype implementation of xR2RML, as well as an experimentation in the context of the Digital Humanities, are presented in Chapter 8.

Figure 9 depicts the object model of xR2RML as an extension of the R2RML model. Two salient modifications can be pointed out. First, the R2RML logical table is extended with the more generic logical source that results of a query executed against the database. The iterator helps deal with hierarchical data models where the natural per-document iteration is not sufficient to achieve all needs. Second, the R2RML term map is extended with the concept of xR2RML reference, the definition of nested term maps to cope with embedded collections, and additional terms types for RDF collections and containers. Further detailed examples are commented in Appendix A, and a **comprehensive specification of the xR2RML mapping language** is provided in Appendix B.

[72] xR2RML stands for “e[x]tended R2RML”

Figure 9: The xR2RML model. xR2RML's extensions of R2RML are highlighted in bold orange.



5.2 Running Example

Along this chapter, we illustrate the xR2RML mapping language using the MongoDB database depicted in Listing 3. Two collections consist of two JSON documents each: a “directors” collection stores documents on movie directors, and a “movies” collection provides information about movies, grouped in per-decade documents.

Collection "directors":

```
{ "name": "Woody Allen",  "directed": ["Manhattan", "Interiors"] }
{ "name": "Wong Kar-wai", "directed": ["2046", "In the Mood for Love"] }
```

Collection "movies":

```
{ "decade": "2000s",
  "movies": [
    {"name": "2046",                 "code": "m2046", "actors": ["T. Leung", "G. Li"]},
    {"name": "In the Mood for Love", "code": "Mood",  "actors": ["M. Cheung"]}
  ]
}
{ "decade": "1970s":,
  "movies": [
    {"name": "Manhattan", "code": "Manh",  "actors": ["Woody Allen", "Diane Keaton"]}
    {"name": "Interiors", "code": "Int01", "actors": ["D. Keaton", "G. Page"]}
  ]
}
```

Listing 3: Content of an example MongoDB database

5.3 Preliminary definitions

An **xR2RML mapping** defines a mapping from a database to RDF; it is represented as an RDF graph called an **xR2RML mapping graph**. An **xR2RML mapping document** is any document written in the Turtle RDF syntax that encodes an xR2RML mapping graph. Any R2RML mapping graph is a valid xR2RML mapping graph (backward compatibility).

An **xR2RML processor,** or **processing engine,** is a system that, given an xR2RML mapping and an input database, provides access to the output RDF dataset. It has access to an execution environment consisting of:

- An **xR2RML mapping document**.
- A **connection** to the input database, used by the xR2RML processor to evaluate queries against the input database. It must be established with sufficient privileges for granting read access to all the database elements (tables, views, documents, objects…) referenced in the xR2RML mapping.
- An optional **reference formulation**: this concept, introduced by RML, specifies a syntax used to reference data elements within results of a query run against the database connection. If it is not provided, it defaults to “column name” in order to ensure backward compatibility with R2RML.
- An optional **query language identifier** identifies which query language shall be used to query the database, in case several languages are supported.
- An optional **base IRI** used in resolving relative IRIs produced by the xR2RML mapping.

5.4 Triples Maps and Logical Sources: from R2RML/RML to xR2RML

To reach its genericity objective, xR2RML must avoid explicitly referring to specific data element reference syntaxes (requirement 1) and specific query languages (requirement 2). Keeping this in mind, we discuss the definitions of the R2RML logical table and RML logical source, in order to come up with a generalized definition of the xR2RML logical source.

5.4.1 R2RML Logical Table vs. RML Logical Source

R2RML logical table: An R2RML logical table is a data set on which a triples map applies: this may be a relational table, an SQL view, or the result of a valid SQL query (property `rr:sqlQuery`).

RML logical source:

- An RML logical source extends the R2RML logical table. It points to a source containing the data to be mapped, denoted by property `rml:source`. In some cases, it brings database connection details (such as the protocol, URL or login provided by a connection string) into the mapping. Whereas this enables several triples maps to refer to difference data sources, it opposes the implicit R2RML idea that such specificities should be kept out of the scope of the mapping. Besides, it is unclear how this property relates to property `rml:query`. The latter is defined in the RML ontology, although it is not described or exemplified in the language specification nor in RML Web pages (as mentioned in specification, September 2014 status). It is only briefly mentioned in an article [Dimou et al., 2013] where authors propose that property `rml:query` subsume properties `rr:sqlQuery` and `rml:xmlQuery`. But this conflicts with requirement 2 since a specific property has to be defined for each supported query language.
- The RML reference formulation concept (property `rml:referenceFormulation`) of an RML logical source names the syntax of data element reference syntaxes (`ql:CSV`, `ql:JSONPath`, `ql:XPath`, `ql:CSS3` and `rr:SQL2008`). This binds the mapping language with a predefined set of syntaxes, which conflicts with requirement 1 and hampers the extensibility to a wider scope of database.

The above discussion underlines that it would not be suitable for xR2RML to extend RML's logical source concept. Instead, the xR2RML logical source extends the R2RML logical table, while relevant RML properties are used or extended separately.

5.4.2 xR2RML Triples Map and Logical Source

The ***xR2RML triples map*** concept extends the R2RML triples map concept by referring to an ***xR2RML logical source*** (property `xrr:logicalSource`) which is the result of executing a request against the database connection. It is either an ***xR2RML base table*** or an ***xR2RML view***.

- The ***xR2RML base table*** extends the concept of *R2RML table or view* to any database returning tabular results: these may be naturally tabular data sources (relational database, NoSQL extensible column store, CSV/TSV files), as well as non-tabular databases providing tabular APIs: for instance,

a graph database equipped with a SPARQL endpoint can return the binding of variables and values of SPARQL Result Sets in tabular format.

- The ***xR2RML view*** represents the result of executing a query string against the input database. It has exactly one `xrr:query` property that extends RML property `rml:query`. Its value is a valid query with regards to the query language supported by the input database. No reference to the type of query language is made within the mapping, such that an xR2RML processor may choose to support any query language deemed appropriate for the considered database, without requiring any extension to the mapping language.

  In addition, an xR2RML view may have one property `rml:iterator` (see section 5.4.4) and any number of `xrr:uniqueRef` properties (see section 5.4.5).

In compliance with R2RML, an xR2RML triples map has exactly one subject map and zero or any number of predicate-object maps. Each predicate-object map may contain at least one predicate map and one object map, and any number of graph maps. We do not go through those details in this chapter, but the interested reader is referred to the language specification in Appendix B.

5.4.3 Reference Formulation

The reference formulation, introduced by RML, specifies a syntax used to reference data elements within the results of a query run against the database connection. To comply with requirement 1, unlike RML, the xR2RML reference formulation is not mentioned in the mapping language but is provided in the xR2RML processing engine environment, typically as a configuration parameter. If it is not provided, it defaults to “column name” to ensure the backward compatibility with R2RML.

It is the responsibility of an xR2RML processor developer to document properly what reference formulations are supported and how they are configured.

5.4.4 Triples Map Iteration Model

In R2RML, the row-based iteration implicitly occurs on a set of rows read from a logical table. xR2RML applies this principle to other row-based systems such as CSV/TSV files and extensible column stores, but also to any tabular result such as a SPARQL result sets. In the context of non-row-based databases, the model is extended to a **document-based iteration model**: a document is a result unit from a result set, whatever the form of such result set. Typically, the document-based iteration applies to a set of JSON documents retrieved from a NoSQL document store, or a set of XML documents retrieved from an XML database. In the case of data sources that do not natively provide iterators over results, for instance a simple file or a Web service returning an XML stream at once, then a single iteration occurs on the whole document.

The document-based iteration model alone may not be sufficient to fulfill all iteration needs. This is particularly true for hierarchical data models such as JSON and XML. To illustrate this issue, let us consider a JSON document from collection “movies” of Listing 3 (some fields are ignored for simplicity):

```
{ "decade": "2000s",
  "movies": [
    {"name": "2046", "code": "m2046"},
    {"name": "In the Mood for Love", "code": "Mood"}
  ]
}
```

We consider the xR2RML mapping graph in Listing 4(a): the JSONPath expression "`$.movies.*.code`" returns two values, hence the subject map produces two terms (one per movie): "m2046", and "Mood". Similarly, the object map produces two terms by applying JSONPath expression "`$.movies.*.name`". This mapping results in mixing up the movie codes and names as shown in Listing 4(b).

(a)
```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.movies.*.code}" ];
  rr:predicateObjectMap [
    rr:predicate dc:title;
    rr:objectMap [ xrr:reference "$.movies.*.name " ]
  ].
```

(b) RDF triples generated:

```
<http://example.org/movie/m2046> dc:title "2046".
<http://example.org/movie/m2046> dc:title "In the Mood for Love".
<http://example.org/movie/Mood>  dc:title "2046".
<http://example.org/movie/Mood>  dc:title "In the Mood for Love".
```

Listing 4: Mixing up RDF terms poduced from a JSON document

To deal with such cases, xR2RML relies on the RML concept of iterator. The previous example is modified as shown in Listing 5(a). The `rml:iterator` property indicates that, within each document retrieved by the query, the triples map iteration should be performed on each movie element rather than on the whole document, thereby producing the expected results in Listing 5(b).

(a)
```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
    rml:iterator "$.movies.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate dc:title;
    rr:objectMap [ xrr:reference "$.name" ]
  ].
```

(b) RDF triples generated:

```
<http://example.org/movie/m2046> dc:title "2046".
<http://example.org/movie/Mood>  dc:title "In the Mood for Love".
```

Listing 5: Using an iterator to avoid mixing up RDF terms

5.4.5 Uniquely Identifying Documents

In Chapter 6, we propose a method to rewrite a SPARQL query into an abstract query under xR2RML mappings. Some optimizations of the produced abstract query require knowing which data elements uniquely identify documents within the database. In a relational database, conventionally, this information is explicitly denoted in the relational schema: primary key and unicity constraints identify the column(s) whose values are unique for all rows of a given table. A query optimization engine can obtain this information simply by exploring the schema.

In schemaless databases however, this information is not made explicit. For instance, in the MongoDB NoSQL document store, each JSON document contains a unique "_id" field, but this field bears no domain semantics, and other fields are generally added to provide a meaningful unique domain identifier. To make such fields explicit, a logical source of type *xR2RML view* may have any number of properties `xrr:uniqueRef`. Each one specifies the xR2RML data element reference of a unique identifier within the documents retrieved by the query (property `xrr:query`). The data element reference is an expression written according to the syntax specified by the reference formulation.

In the example of Listing 6, the `xrr:uniqueRef` property specifies that JSON field "name" is a unique identifier of the documents retrieved from the "directors" collection.

```
<#Directors>
  xrr:logicalSource [
    xrr:query "db.directors.find()";
    xrr:uniqueRef "$.name"
  ];
  rr:subjectMap [ rr:template "http://example.org/dir/{$.name}" ];
  ...
```

Listing 6: Usage of the xrr:uniqueRef property

5.5 Selecting Data Elements from Query Results

5.5.1 Data Element References

R2RML properties `rr:column` and `rr:template` reference column names of a logical table. RML widens R2RML to a larger scope with property `rml:reference` (that extends property `rr:column`): both `rml:reference` and `rr:template` accept data element references expressed according to the reference formulation stated in the RML logical source (column name, XPath, JSONPath…).

xR2RML draws on this idea to meet requirement 1: any syntax can be used transparently in the mapping as a data element reference. As such, xR2RML could simply reuse the `rml:reference` property. However, as we shall see in the next section, it is necessary to extend it to cope with references including custom function calls. Thus, property `xrr:reference` extends `rml:reference`, its object is a data element reference expressed according to the reference formulation provided in the xR2RML processor environment. For instance, the annotation:

```
xrr:reference "$.name"
```

returns the evaluation of JSONPath expression "`$.name`" against the JSON document in the current iteration. Similarly, the definition of property `rr:template` is extended as follows: its object is a template string in which data element references, expressed according to the reference formulation, are enclosed in '{' and '}'. For instance, the annotation:

```
rr:template "http://example.org/movie/{$.code}"
```

concatenates the string `"http://example.org/movie/"` with the result of evaluating JSONPath expression "`$.code`" against the JSON document in the current iteration.

By analogy with R2RML definitions, a term map with an `xrr:reference` property is called a ***reference-valued term map***.

Arguably, an alternative modelling approach would consist in creating a new property `xrr:template` that extends rr:template. RML extended `rr:column` into `rml:reference` in order to avoid confusion: the property name "column" would obviously be inappropriate for an expression that is not a column name. Conversely, the name "template" is generic enough, and RML authors chose to simply amend the property definition, rather than extending it. xR2RML complies with this choice, hence the redefinition of property `rr:template`.

5.5.2 Implement Domain Logic with Custom Functions

RML does not meet requirement 5 (ability to define and use custom functions to implement domain logic). xR2RML addresses this requirement from the following perspective: in real world applications, databases commonly store values written in a data format that they cannot interpret. Let us cite two examples: (i) in key-value stores and in extensible column stores, values are conventionally stored as binary objects whose content is opaque to the system; (ii) in a relational database, an application designer may choose to embed JSON or CSV values for performance concerns or to comply with application design constraints. In such cases, the database query language can retrieve documents or cell values, but it cannot interpret it further. We call such use cases ***mixed content***.

xR2RML extends RML data element references to allow referencing data elements within mixed content. An xR2RML ***mixed-syntax path*** consists of the concatenation of several path expressions, each path being enclosed in a ***syntax path constructor*** that makes the path syntax explicit. Existing constructors are: Column(), CSV(), TSV(), JSONPath() and XPath().

Let us consider the example of a relational table where a text column `NAME` stores JSON-formatted values containing people's first and last names, e.g.:

```
{"First":"John", "Last":"Smith"}
```

Field `FirstName` can be referenced with the mixed-syntax path in the following annotation:

```
xrr:reference "Column(NAME)/JSONPath($.First)"
```

An xR2RML processing engine evaluates a mixed-syntax path from left to right, passing on the result of each path constructor to the next one. In this example, the first path retrieves the value associated with column `NAME`. The value is passed on to the next path constructor that evaluates JSONPath expression "`$.First`". The resulting value is finally translated into an RDF term according to the current term map definition.

xR2RML states that both properties `xrr:reference` and `rr:template` accept either simple data element references (previous section) or mixed-syntax path expressions.

Arguably, this definition partially contradicts requirement 2. Indeed, an explicit list of constructors CSV(), TSV(), JSONPath() and XPath(), is embedded in the mapping language definition, thereby limiting the extensibility to other data formats. In section 5.8 we suggest perspectives to overcome this limitation.

5.6 Producing RDF Terms and (Nested) RDF Collections/Containers

In a row-based logical source, a valid column name reference returns at most one value during each triples map iteration. Subsequently, an R2RML term map generates zero or one RDF term per iteration. By contrast, JSONPath and XPath expressions provided as values of properties `xrr:reference` and `rr:template` may generate multiple values. Typically, XPath expression "`//movie/name`" would return all `<name>` elements of all `<movie>` elements. Therefore, unlike R2RML, xR2RML reference-valued and template-valued term maps can return multiple RDF terms at once (during one iteration). This difference entails the definition of two strategies with regards to how xR2RML triples maps combine RDF terms to build triples: the *product strategy*, and the *collection/container strategy*.

Product strategy. During each iteration of an xR2RML triples map, triples are generated as the product between RDF terms produced by the subject map and each predicate-object pair. Similarly, predicate-object pairs result of the product between RDF terms produced by the predicate maps and RDF terms produced by the object maps of each predicate-object map. Like any other term map, a graph map may also produce multiple terms. The product strategy equally applies in that case, thereby producing RDF triples simultaneously in the target graphs corresponding to the multiple RDF terms produced by the graph map.

In Listing 7(a), triples map `<#Movies>` relates each movie to one actor starring in that movie. Actors are produced by the xR2RML reference "$.actors.*" that generates multiple terms. Listing 7(b) depicts the RDF triples generated when applying this triples map to the documents in Listing 3.

Collection vs. container strategy. Multiple values returned by properties `xrr:reference` and `rr:template` are combined into an RDF collection or container. This is achieved using new xR2RML values of the `rr:termType` property: a term map with term type `xrr:RdfList` generates an RDF term of type `rdf:List`, term type `xrr:RdfSeq` corresponds to `rdf:Seq`, `xrr:RdfBag` to `rdf:Bag` and `xrr:RdfAlt` to `rdf:Alt`. In Listing 8(a), instead of generating multiple triples relating each movie to one actor starring in that movie, triples map `<#Movies>` relates each movie to a bag of actors. Listing 8(b) depicts the RDF triples generated when applying this triples map to the documents in Listing 3.

(a)
```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
    rml:iterator "$.movies.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:starring;
    rr:objectMap [ xrr:reference "$.actors.*" ]
  ].
```

(b) RDF triples generated:

```
<http://example.org/movie/m2046> ex:starring "Tony Leung.
<http://example.org/movie/m2046> ex:starring "Gong Li".
```

Listing 7: Mapping of a list of elements to multiple triples

(a)
```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
    rml:iterator "$.movies.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:starring;
    rr:objectMap [
      rr:termType xrr:RdfBag;
      xrr:reference "$.actors.*"
    ]
  ].
```

(b) RDF triples generated:

```
<http://example.org/movie/m2046> ex:starring [
    a rdf:Bag;
    rdf:_1 "Tony Leung";
    rdf:_2 "Gong Li"
].
```

Listing 8: Mapping of a list of elements to an RDF bag

At this point, two important needs must still be addressed in the collection/container strategy:

(i) like in a regular term map, it must be possible to assign a *term type*, *language tag* or *data type* to the members of an RDF collection or container; and

(ii) it must be possible to nest any number of RDF collections and containers inside each-other.

Both needs are fulfilled using ***xR2RML Nested Term Maps***. A nested term map (property `xrr:nestedTermMap`) very much resembles a regular term map, with the exception that it can be defined only inside a term map that produces an RDF collection or container. In a column-valued or reference-valued term map, a nested term map describes how to translate values read from the logical source into RDF terms, by specifying optional properties `rr:termType`, `rr:language` and `rr:datatype`. Similarly,

in a template-valued term map, a nested term map applies to values produced by applying the template string to input values.

In Listing 9(a), triples map `<#Movie>` uses a nested term map to assign the `xsd:string` datatype to actor names. The result is illustrated in Listing 9(b).

(a)

```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
    rml:iterator "$.movies.*";
  ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:starring;
    rr:objectMap [
      rr:termType xrr:RdfBag;
      xrr:reference "$.actors.*";
      xrr:nestedTermMap [ rr:datatype xsd:string ]
    ]
  ].
```

(b) RDF triples generated:

```
<http://example.org/movie/m2046> ex:starring [
    a rdf:Bag;
    rdf:_1 "Tony Leung"^^xsd:string;
    rdf:_2 "Gong Li"^^xsd:string
].
```

Listing 9: Assigning a data type to members of an RDF collection/container

Input data

```
{ "teams": [ ["John", "Paul"] , ["Cathy", "Ed"] ] }
```

Term map

```
[] rr:objectMap [
     xrr:reference "$.teams.*";
     rr:termType xrr:RdfSeq;      # Represents "teams" as an rdf:Seq
     xrr:nestedTermMap [          # Describes elements of the rdf:Seq

       rr:termType xrr:RdfList;   # Represents each team as an rdf:List
       xrr:nestedTermMap [        # Describes elements of the inner rdf:List

          rr:template "Player {$.*}";
          rr:termType rr:Literal; # Types each player as literal
          rr:language "en";       # Adds language "en"
       ];
     ];
  ];
```

Generated RDF terms

```
[ a rdf:Seq;
  rdf:_1 ("Player John"@en "Player Paul"@en);
  rdf:_2 ("Player Cathy"@en "Player Ed"@en);
]
```

Listing 10: Assigning a data type to members of an RDF collection/container

Furthermore, properties `xrr:reference` and `rr:template` can be used within the context of a nested term map in order to recursively parse nested structured values and produce nested RDF collections and containers. In Listing 10, we exemplify an object map with two nested term maps nested within each other. The expected result in an RDF sequence of RDF lists of literals tagged with the language tag "`@en`".

5.7 Cross-Referenced Logical Sources

Cross-references allow modelling relationships between different logical entities of a database, or across several databases. Translated to RDF, a cross-referenced logical entity may be mapped alternatively as the subject or the object of RDF triples.

In R2RML, a *referencing object map* uses IRIs produced by the subject map of a triples map (called the *parent* triples map) as the objects of triples produced by another triples map (the *child* triples map). In case both triples maps do not share the same logical table, a join query must be performed. A join condition (property `rr:joinCondition`) names the columns from the parent and child triples maps, that must be joined (properties `rr:parent` and `rr:child`).

xR2RML extends R2RML referencing object maps in two ways that we describe in the subsequent sections.

5.7.1 Join query with multi-valued data element references

xR2RML extends the definition of properties `rr:child` and `rr:parent` such that their object can now be data element references expressed in the reference formulation, possibly including mixed-syntax paths. As underlined in section 5.6, such data element references may produce multiple terms. Consequently, the equivalent join query of a referencing object map must deal with multi-valued child and parent references. More precisely, a join condition between two multi-valued references should be satisfied if at least one data element of the child reference matches one data element of the parent reference. This is formalized in Definition 1 using an SQL-like syntax and first order logic for the description of WHERE conditions.

Definition 1. Equivalent join query

If an xR2RML referencing object map has at least one join condition, its equivalent join query is defined as follows:

SELECT * FROM ({*child-query*}) AS **child**, ({*parent-query*}) AS **parent**
WHERE
$\exists c_1 \in$ eval(**child**, {*child-ref*$_1$}), $\exists p_1 \in$ eval(**parent**, {*parent-ref*$_1$}), $c_1 = p_1$
AND
$\exists c_2 \in$ eval(**child**, {*child-ref*$_2$}), $\exists p_2 \in$ eval(**parent**, {*parent-ref*$_2$}), $c_2 = p_2$
AND ...

where {*child-ref*$_n$} and {*parent-ref*$_n$} are the child and parent references of the n^{th} join condition, and eval(source, {ref}) is the result of evaluating the data element reference "{ref}" against data "source".

Listing 11 depicts an example: in triples map `<#Directors>`, the object map uses movie IRIs generated by parent triples map `<#Movies>`. When processing director “Wong Kar-wai”, the child reference (`$.directed.*`) returns values “2046” and “In the Mood for Love”, while the parent reference (`$.name`) returns a single movie name for each document of the “movie” collection. The join condition is satisfied if the parent reference returns one of “2046” or “In the Mood for Love”.

```
<#Movies>
  xrr:logicalSource [
    xrr:query "db.movies.find({decade:{$exists:true}})";
    rml:iterator "$.movies.*" ];
  rr:subjectMap [ rr:template "http://example.org/movie/{$.code}" ];
  rr:predicateObjectMap [
    rr:predicate ex:starring;
    rr:objectMap [ xrr:reference "$.actors.*" ]
  ].

<#Directors>
  xrr:logicalSource [ xrr:query "db.directors.find()" ];
  rr:subjectMap [ rr:template "http://example.org/dir/{$.name}" ];
  rr:predicateObjectMap [
    rr:predicate ex:directed;
    rr:objectMap [
      rr:parentTriplesMap <#Movies>;
      rr:joinCondition [
        rr:child "$.directed.*";
        rr:parent "$.name" ]
  ] ].
```

Listing 11: Cross-reference logical entities with a referencing object map

According to Definition 1, the join query entailed from triples map `<#Directors>` can be written as follows:

```
SELECT * FROM ("db.directors.find()") as child,
              ("db.movies.find()".iterator("$.movies.*")) as parent
WHERE ∃ c ∈ eval(parent, $.name), ∃ p ∈ eval(child, $.directed.*), c = p
```

Below, we rewrite the result of this join query in tabular form; matching child and parent values are highlighted in red:

child (directors)		parent (movies)		
name	directed	name	code	actors
Woody Allen	["Manhattan", "Interiors"]	Manhattan	Manh	["Woody Allen", "Diane Keaton"]
Woody Allen	["Manhattan", "Interiors"]	Interiors	Int01	["D. Keaton", "G. Page"]
Wong Kar-wai	["2046", "In the Mood for Love"]	2046	m2046	["T. Leung", "G. Li"]
Wong Kar-wai	["2046", "In the Mood for Love"]	In the Mood for Love	Mood	["M. Cheung"]

On each line, the object map returns the RDF term generated by `<#Movies>`' subject map (the subject map of the parent triples map), using column "code". We can deduce the RDF triples ultimately produced:

```
<http://example.org/dir/Woody%20Allen>  ex:directed
     <http://example.org/movie/Manh>, <http://example.org/movie/Int01>.
<http://example.org/dir/Wong%20Kar-wai> ex:directed
     <http://example.org/movie/m2046>, <http://example.org/movie/Mood>.
```

5.7.2 From a One-to-Many Relationship to an RDF Collection/Container

In addition, xR2RML allows grouping the objects produced by a referencing object map in an RDF collection or container, instead of being the objects of multiple triples. To do so, an xR2RML referencing object map may have a `rr:termType` property with value `xrr:RdfList`, `xrr:RdfSeq`, `xrr:RdfBag` or `xrr:RdfAlt`. Results of the join query pertaining to the same subject are grouped together.

As an illustration, we amend triples map `<#Directors>` of Listing 11 as follows:

```
rr:objectMap [
  rr:parentTriplesMap <#Movies>;
  rr:termType xrr:RdfBag;
  rr:joinCondition [
    rr:child "$.directed.*";
    rr:parent "$.name" ]
] ].
```

The `rr:termType` property now instructs the processor to group movie IRIs pertaining to the same director into an RDF bag. In turn, we obtain the following RDF triples:

```
<http://example.org/dir/Woody%20Allen> ex:directed [ a rdf:Bag;
     rdf:_1 <http://example.org/movie/Manh>;
     rdf:_2 <http://example.org/movie/Int01>
].
<http://example.org/dir/Wong%20Kar-wai> ex:directed [ a rdf:Bag;
     rdf:_1 <http://example.org/movie/m2046>;
     rdf:_2 <http://example.org/movie/Mood>
].
```

An interesting consequence of this use case is the ability, with a regular relational database, to build an RDF collection or container reflecting a one-to-many relation between tables.

5.8 Perspectives

***Direct Mapping*.** Like R2RML, xR2RML assumes that well-defined domain ontologies exist beforehand, whereof classes and properties are used to translate a data source into RDF triples. Commonly though, no appropriate ontology is available up front, and people willing to expose an existing database in RDF have to define an ontology or extend existing ones. To bootstrap the process, the direct mapping approach proposes a straightforward method to come up with an ad-hoc ontology that roughly reflects the database schema. Substantial works have been achieved to apply the direct mapping in the context of relational databases [Sequeda et al., 2011; Arenas et al., 2012] by exploring the relational database schema. Similar works attempt to transform an XML Schema (XSD) to OWL [Bohring & Auer, 2005; Bedini et al., 2011]. Some direct mapping principles can also be applied regardless of any schema. For instance, TARQL [Cyganiak, 2013] simply translates each JSON field name into an ad-hoc predicate name, but no class name is ever created. In the latter, the ontology emerges as the data translation process is running, whereas in the other aforementioned approaches it can be generated in a first step, using schema metadata (relational schema, XSD).

R2RML implementations often come with a tool to automatically generate an R2RML direct mapping from the relational schema (e.g. MIRROR [de Medeiros et al., 2015] and D2RQ). In line with this idea, we could automatically derive an xR2RML direct mapping from various databases, as long as they exhibit a data schema (table/column names, XSD, DTD, JSON schema[73], etc.), thus implicitly creating class and predicate names of an ad-hoc ontology.

***Automate Mappings Generation*.** At this point, we have considered that the writing of xR2RML mappings is a manual process involving people having sufficient expertise with respect to the database schema, the domain ontologies, and the xR2RML mapping language. However, how to automate the generation of xR2RML mappings may become an issue when it comes to map large and/or complex schemas. There exists significant work related to schema mapping and matching [Shvaiko & Euzenat, 2005]. For instance, Karma [Knoblock et al., 2012] semi-automatically maps structured data sources to existing domain ontologies. It produces a Global-and-Local-As-View mapping that can be used to translate the data into RDF, and the authors suggest that Karma could easily export mapping rules as an R2RML mapping graph.

xR2RML does not address the question of how mappings are written, but it could be complementary of approaches like Karma. Most likely, it should be possible to draw on existing techniques to discover mappings between a non-relational database and domain ontologies, and export the result as an xR2RML mapping graph.

[73] JSON Schema: http://json-schema.org/

Custom functions with CSVW/R2RML-F. In Chapter 4, we identified R2RML-F and CSVW as interesting candidates to implement domain logic by means of custom functions (requirement 6). Below we sketch how xR2RML could rely on CSVW and R2RML-F.

R2RML-F [Debruyne & O'Sullivan, 2016] proposes to incorporate domain logic in R2RML mappings by means of custom functions. First, functions are described as an RDF resource with a function name and an ECMAScript function body. For instance, the function below takes two parameters and returns their product:

```
<#Multiply>
   rrf:functionName "multiply" ;
   rrf:functionBody """
      function multiply(var1, var2) { return var1 * var2 ; }
   """.
```

Second, R2RML-F defines the concept of *function-call-valued term map*, to complement existing *constant-valued, column-valued* and *template-valued* R2RML term maps. A *function-call-valued* term map is a term map that can produce RDF terms by invoking a function. Arguments of the function are themselves regular term maps passed as an RDF list. The example object map below calls function `<#Multiply>` to compute the product of the value in column `COL_NAME` with 12.

```
...
rr:objectMap [
  rrf:functionCall [
    rrf:function <#Multiply>;
    rrf:parameterBindings (
      [ rr:constant "12"^^xsd:integer ]
      [ rr:column "COLNAME" ]
    )
  ]
];
```

Meanwhile, CSVW has defined a somewhat more generic method describing custom functions as part of the W3C *Metadata Vocabulary for Tabular Data* [Pollock et al., 2015], and general enough to apply beyond the scope of tabular data. A CSVW *transformation* refers not only to scripts but also to templates. The script/template format is annotated with its media type[74], or any URL describing the format if no media type is defined. Similarly, the output of the script/template is annotated with a media type. In the example below, the template at URL "templates/ical.txt" follows the Mustache template format[75] (property `scriptFormat`) and it produces a result with the calendar media type (property `targetFormat`).

```
{ "url": "templates/ical.txt",
  "titles": "iCalendar",
  "targetFormat": "http://www.iana.org/assignments/media-types/text/calendar",
  "scriptFormat": "https://mustache.github.io/",
  "source": "json"
}
```

[74] Media types: http://www.iana.org/assignments/media-types/media-types.xhtml
[75] Mustache templates: http://mustache.github.io/

While CSVW defines a quite generic way to refer to scripts and formats, R2RML-F proposes a syntax that already complies with the R2RML framework. Quite naturally, we can figure out an extension of xR2RML including R2RML-F's function-call term maps and CSVW's transformations.

We illustrate this proposition in Listing 12. A function `<#Multiply>` is defined with CSVW properties and the R2RML-F's `rrf:functionName` property (lines 4 to 9). Assuming the context of a JSON data source, the object map calls function `<#Multiply>` (line 16), passing two arguments: the first is a constant term map (line 18), the second is an xR2RML reference-valued term map that uses a JSONPath expression to retrieve a value from the source document (line 19).

```
1  @prefix csvw: <http://www.w3.org/ns/csvw#>.
2  @prefix rrf: <http://kdeg.scss.tcd.ie/ns/rrf#>.
3
4  <#Multiply>
5     rrf:functionName "multiply";
6     csvw:url "scripts/multiply.js";
7     csvw:targetFormat xsd:integer;
8     csvw:scriptFormat "http://example.org/media-types/multiply".
9
10 <#SomeTriplesMap>
11    xrr:logicalSource [ ... ];
12    rr:predicateObjectMap [
13       ...
14       rr:objectMap [
15          rrf:functionCall [
16             rrf:function <#Multiply>;
17             rrf:parameterBindings (
18                [ rr:constant "12"^^xsd:integer ];
19                [ xrr:reference "$.fieldName" ];
20             )
21          ]
22       ]
23    ].
```

Listing 12: Extension of xR2RML with custom function using the formalisms of R2RML-F and CSVW

5.9 Conclusion

In Chapter 4, we identified a set of requirements that a generalized mapping language should meet in order to enable the mapping of various databases to RDF, by flexibly adapting to heterogeneous query languages and data models. In this chapter, we proposed the definition of such a generalized mapping language, namely xR2RML. Below we review how xR2RML meets each of the aforementioned requirements.

Requirement 1: Open set of data element reference syntaxes. The mapping language must remain free from any explicit reference to specific syntaxes.

xR2RML properties `xrr:reference` and `rr:template` allow data element references in various syntaxes. This syntax, called the *reference formulation,* is provided in the xR2RML processing engine environment. No reference to any reference formulation is made within the mapping language, such that an xR2RML processor may choose to support any syntax deemed appropriate for the considered database (sections 5.4.2 and 5.4.3).

Requirement 2: Open set of database query languages. The mapping language must remain free from any explicit reference to specific query languages.

An xR2RML view represents the result of executing a query string against the input database. The `xrr:query` property is a valid expression with respect to the query language supported by the input database. No reference to any query language is made within the mapping language, such that an xR2RML processor may choose to support any query language deemed appropriate for the considered database (sections 5.4.2 and 5.4.4).

Requirement 3: Generate RDF collections (list) and containers (bag, seq, alt) to preserve the semantics of collections and one-to-many relationships.

Term maps produce RDF terms whose type is given by R2RML property `rr:termType`. The range thereof is extended by xR2RML with values `xrr:RdfList`, `xrr:RdfSeq`, `xrr:RdfBag` and `xrr:RdfAlt`. A term map with one of these term types produces an RDF collection or container resource that groups the multiple values produced by the reference or template. Nested collections and containers are supported (section 5.6).

Additionally, those term types may be used to turn a one-to-many relationship of the database schema into an RDF collection or container (section 5.7.2).

Requirement 4: Allow the description of join queries to retrieve cross-referenced logical entities.

xR2RML extends the definition of properties `rr:child` and `rr:parent` such that their object be data element references expressed in the reference formulation. Insofar as data element references may be multi-valued (they may produce multiple terms), Definition 1 describes how to write the equivalent join query of a referencing object map when child and/or parent references are multi-valued (section 5.7.1).

Requirement 5: Enable custom functions implementing domain logic.	
✓ partially	xR2RML data element references allow referencing data elements within mixed content (when the database schema contains elements whose content is opaque to the database), or in a format the database cannot natively interpret (typically when JSON objects are embedded in extensible column stores, or when a column of a relational table stores content formatted in XML, JSON etc.) (section 5.5.2). Yet, this solution does not provide the flexibility that may be required in certain use cases; we discuss this issue in the perspectives (5.8).

Requirement 6: Allow assigning generated RDF triples to a named graph or to the default graph.	
✓	xR2RML relies on the concept of R2RML graph maps, with the difference that, like any other term map, an xR2RML graph map uses the reference formulation to refer to data elements within query results.

To achieve the expected flexibility, xR2RML comes with features that are applicable independently of the type of database used. Yet, not all features should be applied regardless of the context. We have shown that join conditions of referencing object maps entail join queries. While relational databases are optimized to support joins very efficiently, NoSQL document or column stores are optimized for denormalized data models where redundancy is the rule and not the exception, and thus hardly support joins. Consequently, thoughtlessly mapping such NoSQL stores to RDF using join conditions may entail very poor performances. In other words, because the language makes a mapping possible does not mean that it should be applied regardless of the database type, its data model and query capabilities. Mapping designers should be aware of the database machinery in order to map big datasets to RDF efficiently.

In Table 3, we remind and extend Table 2 by including a last line for xR2RML.

Table 3: Comparison of mapping languages (including xR2RML) against the requirements for a generalized mapping language

Requirement / Language	1: Open set of data element reference syntaxes	2: Open set of query languages	3: Generate RDF collections and containers	4: Follow cross-references	5: Custom functions	6: Named graphs	No. of reqmts. met
R_2O [Barrasa et al., 2004]	✗ (Column)	✗ (SQL)	✗	✓	✓ (proprietary)	✓	3
D2RQ [Bizer & Cyganiak, 2006]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
SML [Stadler et al., 2015]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
R2RML [Das et al., 2012]	✗ (Column)	✗ (SQL)	✗	✓	✗ (SQL)	✓	2
R2RML-F [Debruyne & O'Sullivan, 2016]	✗ (Column)	✗ (SQL)	✗	✓	✓ (ECMAscript)	✓	3
RML [Dimou et al., 2014a]	✓	✗	✗	✓	✗	✓	3
XSPARQL [Bischof et al., 2012]	✗ (XPath)	✗ (XQuery, SPARQL)	✓ (XQuery)	✓ (XQuery)	✓ (XPath)	✗	3
SPARQL2XQuery [Bikakis et al., 2013]	✗ (XPath)	✗ (XQuery)	✗	✓ (XQuery)	✓ (XPath)	✓	3
CSVW [Tandy et al., 2015]	✗ (Column)	✗	✓ (rdf:List only)	✓	✓	✗	3
XLWrap [Langegger & Wöss, 2009]	✗ (Row/Col.)	✗	✗	✓	✓ (OpenOffice Calc)	✗	2
xR2RML	✓	✓	✓	✓	✓ (proprietary)	✓	6

Chapter 6. Mapping-Based SPARQL Access to Heterogeneous Databases

6.1 Introduction

In Chapter 5, we defined the xR2RML generalized mapping language, meant to enable the mapping of heterogeneous databases to RDF. By fostering the translation of legacy data sets into RDF, xR2RML proposes to be a building block of RDF-based data integration systems, and in particular, it intends to amplify the Web-scale data integration trend that progressively populates the Web of Data. Two approaches generally apply when it comes to translate legacy data into RDF. In the graph materialization approach, legacy data is all translated at once into an equivalent RDF graph. Conversely, in the dynamic access approach, data is accessed on-the-fly as a virtual RDF graph using the SPARQL query language. Although the materialization is of interest in some contexts, it is hardly a one-fits-all solution in practice. In particular, dynamic access scales better to large data sets and guarantees data freshness.

In this matter, it shall be necessary to develop SPARQL access methods for heterogeneous databases that vary greatly in terms of query languages: typically, relational and XML databases support joins, nested queries and string manipulation functions, but this is hardly the case of NoSQL systems (apart from graph databases) such as the MongoDB or CouchDB document stores, that typically trade off query language expressiveness for scalability and performance. For instance, MongoDB does not support joins, and only supports nested queries under strong restrictions. Thus, to avoid defining yet another SPARQL translation method for each and every target database query language, we introduce a two-phase approach:

1. The first phase enacts the database-independent steps. Given a set of xR2RML mappings of the target database to RDF, a SPARQL query is translated into a pivot abstract query by matching SPARQL graph patterns with relevant xR2RML mappings. Given that xR2RML transparently applies to most types of databases, this step is independent of any target database specificity.
2. Conversely, the second phase enacts the steps that are specifically dependent on the target database: the abstract query is translated into the target database query language, taking into account the specific database query capabilities.

This chapter specifically focuses on phase 1. Leveraging previous works on R2RML-based SPARQL-to-SQL translation, we define a pivot abstract query language and a method to translate a SPARQL query into an abstract query under xR2RML mappings. The method determines a reduced set of mappings that match each of the SPARQL triple patterns, while taking into account SPARQL filters as well as join constraints implied by shared variables and cross-references denoted in the mappings. Query optimization techniques are applied to the abstract query in order to alleviate the work required in the second step.

Chapter 7 demonstrates the effectiveness of the method in the concrete case of the MongoDB NoSQL document store.

6.1.1 Normalization of xR2RML Mappings

A standard principle when rewriting a SPARQL query based on mappings is to identify which mappings are good candidates for each triple pattern in the SPARQL graph pattern. The most specific the mappings are, the most accurate the matching will be. For instance, if a mapping produces triples with two possible predicates, then triple patterns with either one or the other predicate will match this mapping. If this mapping can be split into two mappings, one for each predicate, then it is likely that the matching will be more accurate.

In xR2RML and R2RML, mappings (triples maps) may contain any number of predicate-object maps, and each predicate-object map may contain any number (≥1) of predicate maps and object maps. Therefore, a triples map with multiple predicate-object maps (and/or multiple predicate and object maps) may generate varying types of RDF triple, and result in the coarse matching of this triples map with triple patterns.

This issue has been addressed in R2RML-based rewriting approaches. Rodríguez-Muro and Rezk [Rodríguez-Muro & Rezk, 2015] propose an algorithm to *normalize* R2RML mappings, that is, rewrite the mapping graph so that a triples map contain at most one predicate-object map, each having exactly one predicate map and one object map. Although they do not explicitly mention it, authors of [Unbehauen et al., 2013a; Priyatna et al., 2014] do the same assumption.

Furthermore, the R2RML `rr:class` property introduces a specific way of producing triples such as "`<A> rdf:type <B>`". Rodríguez-Muro and Rezk propose to replace any `rr:class` property by an equivalent predicate-object map: `[rr:predicate rdf:type; rr:object <B>.]`. This allows for the definition of a rewriting method consistently dealing with all kinds of triple patterns, may they have the `rdf:type` property or any other property.

We comply with both of the aforementioned propositions as they apply to R2RML and xR2RML alike. This is summarized is Definition 2.

> ***Definition 2. Normalized xR2RML Triples Map***
>
> An xR2RML triples map is said to be normalized when:
>
> (i) It contains at most one predicate-object map;
>
> (ii) The predicate-object map contains exactly one predicate map and one object map;
>
> *(iii)* Its logical source definition does not use the `rr:class` property, instead a regular predicate-object map is used to generate RDF triples with a constant predicate `rdf:type` and a constant object.

In the rest of this chapter, we only consider normalized xR2RML triples map.

6.1.2 Running Example

To illustrate the description of our method, we define a running example to which we shall refer all along this chapter. To keep focused on the query translation challenges, and for the sake of clarity, the running example does not involve iterators (`rml:iterator`) nor xR2RML mixed syntax paths.

Let us consider a MongoDB database with two collections “staff” and “departments” depicted in Listing 13. Collection “departments” lists the departments within a company, described by a department code and a list of members. Members are given by their name and age. Collection “staff” lists people by their name (that may be either field “familyname” or “lastname”), and provides a list of departments that they manage, if any, in array field “manages”.

Collection "staff":

```
{ "familyname":"Underwood", "manages":["Sales"] },
{ "lastname":"Dunbar",      "manages":["R&D", "Human Resources"] },
{ "lastname":"Sharp",       "manages":["Support", "Business Dev"] }
```

Collection "departments":

```
{ "dept":"Sales",           "code":"sa",
  "members":[
     {"name":"P. Russo",    "age":28},
     {"name":"J. Mendez",   "age":43}
]}
{ "dept":"R&D",             "code":"rd",
  "members": [
     {"name":"J. Smith",    "age":32},
     {"name":"D. Duke",     "age":23}
]}
{ "dept":"Human Resources", "code":"hr",
  "members": [
     {"name":"R. Posner",   "age":46},
     {"name":"D. Stamper",  "age":38}
]}
{ "dept":"Business Dev",    "code":"bdev",
  "members": [
     {"name":"R. Danton",   "age":36},
     {"name":"E. Meetchum", "age":34}
]}
```

Listing 13: Example MongoDB database

Let us consider the xR2RML mapping graph in Listing 14, consisting of two triples maps <#Staff> and <#Departments>. For the sake of simplicity, the queries in both triples maps retrieve all documents of the collection with no other query filter: the logical source in triples map <#Staff> provides a MongoDB query `db.staff.find({})` that retrieves all documents in collection “staff”; similarly, the query in the logical source of triples map <#Departments> retrieves all documents in collection “departments”. Triples map <#Staff> has a referencing object map whose parent triples map is <#Departments>.

Triples map <#Departments> generates triples with predicate `ex:hasSeniorMember` for each member of the department who is 40 years old or more.

```
<#Departments>
   xrr:logicalSource [
      xrr:query "db.departments.find({})";
      xrr:uniqueRef "$.code"
   ];
   rr:subjectMap [ rr:template "http://example.org/dept/{$.code}" ];
   rr:predicateObjectMap [
      rr:predicate ex:hasSeniorMember;
      rr:objectMap [ xrr:reference "$.members[?(@.age >= 40)].name" ]
   ].

<#Staff>
   xrr:logicalSource [ xrr:query "db.staff.find({})" ];
   rr:subjectMap [ rr:template "http://example.org/staff/{$['lastname','familyname']}" ];
   rr:predicateObjectMap [
      rr:predicate ex:manages;
      rr:objectMap [
         rr:parentTriplesMap <#Departments>;
         rr:joinCondition [
            rr:child "$.manages.*";
            rr:parent "$.dept"
         ]
      ]
   ].
```

Listing 14: xR2RML Example Mapping Graph

Finally, we consider the SPARQL query in Listing 15, that we shall use throughout this chapter to illustrate the method. It retrieves senior members of departments managed by a person named "Dunbar". The query consists of one basic graph pattern composed of two triple patterns tp_1 and tp_2.

```
SELECT ?senior WHERE {
    <http://example.org/staff/Dunbar> ex:manages ?dept. # tp1
    ?dept ex:hasSeniorMember ?senior.                   # tp2
}
```

Listing 15: Example SPARQL Query

6.2 Previous Works Related to SPARQL Rewriting

During the last decade, several works proposed to achieve SPARQL access to relational data, either in the context of RDB-based RDF stores [Chebotko et al., 2009; Sequeda & Miranker, 2013; Elliott et al., 2009] or using arbitrary relational schemas [Bizer & Cyganiak, 2006; Unbehauen et al., 2013a; Priyatna et al., 2014; Rodríguez-Muro & Rezk, 2015]. Note that all of these methods consider SPARQL 1.0 [Harris & Seaborne, 2008]; to our knowledge, no rewriting method supports SPARQL 1.1 [Harris & Seaborne, 2013: 1] as of today.

These methods harness the ability of SQL to support joins, unions, nested queries and various string manipulation functions. Typically, a conjunction of two basic graph patterns (BGP) results in the inner

join of their respective translations; their union results in an SQL UNION ALL clause; the SPARQL OPTIONAL clause between two BGPs results in a left outer join, and a SPARQL FILTER results in an encapsulating SQL SELECT WHERE clause.

Chebotko's algorithm [Chebotko et al., 2009] focuses on RDB-based triple stores. Priyatna et al. [Priyatna et al., 2014] extend it to support custom R2RML mappings and apply several query optimizations. Two limitations can be underlined though:

(i) R2RML triples maps must have constant predicate maps, i.e. the predicate term of the generated RDF triples cannot be built from database values;

(ii) Triple patterns are considered and translated independently of each other, even when they share SPARQL variables. The resulting SQL query embeds unnecessarily complexity that is taken care of later on, in the SQL query optimization step. Hence, the optimization is dependent on the target database language, and can hardly be generalized. In our attempt to rewrite SPARQL queries in the general case, such optimization shall be performed as early as possible, regardless of the target database capabilities.

Unbehauen et al. [Unbehauen et al., 2013a] define the concept of compatibility between the RDF terms of a SPARQL triple pattern and R2RML term maps. This more general approach effectively manages variable predicate maps, which clears the first aforementioned limitation. Furthermore, they reduce the number of candidate triples maps for each triple pattern by pre-checking join constraints implied by shared variables. This clears the second aforementioned limitation. Yet again, two limitations can be noticed:

(iii) R2RML referencing object maps are not considered, therefore joins implied by shared variables are dealt with but joins declared in the R2RML mapping graph are ignored.

(iv) The rewriting process maps each term map to a set of columns, called *column group*, which enables filter, join and data type compatibility checks. This leverages SQL capabilities (CASE, CAST, string concatenation, etc.), making it hardly applicable out of the scope of SQL-based systems.

Rodríguez-Muro and Rezk [Rodríguez-Muro & Rezk, 2015] propose a different approach. They extend the *ontop* Ontology-Based Data Access system, to support R2RML mappings. A SPARQL query and an R2RML mapping graph are translated into a Datalog program. This formal representation is used to combine and apply optimization techniques from logic programming and SQL querying. The optimized program is then translated into an executable SQL query.

Finally, it occurs that the SPARQL-to-SQL methods reviewed above are tailored to the expressiveness of the target query language: SQL specificities are woven into the translation method itself, which undermines the ability to use such methods beyond relational databases. A similar situation is observed with respect to XML databases. For instance, SPARQL2XQuery [Bikakis et al., 2015] relies on the ability of XQuery to support joins, nested queries and complex filtering. Typically, a SPARQL FILTER is translated into an encapsulating For-Let-Where XQuery clause. The rich expressiveness of SQL and XQuery makes it possible to translate a SPARQL query into a single, possibly deeply nested, target query, whose semantics is strictly equivalent to that of the SPARQL query. Query optimization issues may be addressed early in the translation process, later on at the level of the produced target query, or they may simply be delegated to the target database optimization engine.

Therefore, in order to address a large scope of target databases, we must generalize those approaches to make them independent of any target query language. This will be the object of the subsequent sections.

6.3 Rewriting a SPARQL Query into an Abstract Query under Normalized xR2RML Mappings

Section 6.2 has shown that SPARQL rewriting methods for SQL or XQuery rely on the expressiveness of the target query language to perform a semantics-preserving translation: a SPARQL query is translated into a single equivalent target query. In the general case however (beyond SQL and XQuery), the target query language may not support joins, unions and/or sub-queries. To tackle this issue, the first translation phase enacts the database-independent steps. We rely on and extend the R2RML-based SPARQL rewriting approaches reviewed in section 6.2, while taking care of avoiding the limitations highlighted. Below, we do not consider the specific types of SPARQL queries (SELECT, ASK, DESCRIBE, etc.) but we focus on rewriting SPARQL graph patterns. The translation of a SPARQL graph pattern into an abstract query consists of four steps sketched in Figure 10.

d

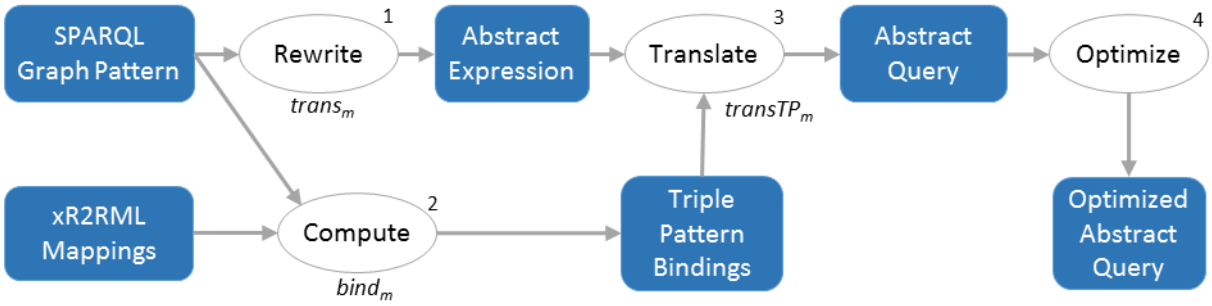


Figure 10: Translation of a SPARQL 1.0 graph pattern into an optimized abstract query

1. A SPARQL 1.0 graph pattern is decomposed and rewritten into an abstract expression exhibiting operators from the abstract query language (function $trans_m$, §6.4).
2. Then, we identify a set of xR2RML mappings (triples maps) likely to generate RDF triples that match each triple pattern of the SPARQL graph pattern (function $bind_m$, §6.5).
3. Each triple pattern is translated into a sub-query consisting of operators of the abstract query language and atomic abstract queries, under the set of xR2RML triples maps identified in step 2 (function $transTP_m$, §6.6).
4. Finally, we perform several types of optimization on the resulting abstract query, *e.g.* by removing self-joins or self-unions (§6.7).

Abstract Query Language. Our pivot abstract query language complies with the grammar defined in Definition 3, that directly derives from the syntax and semantics of SPARQL [Pérez et al., 2009]. The language keeps the names of several SPARQL operators (UNION, LIMIT, FILTER) and prefers the SQL terms INNER JOIN ON and LEFT OUTER JOIN ON to refer to join operations more explicitly. Like in the case of SPARQL, an abstract query can be represented as a tree. However, the abstract query language

differs from SPARQL in that the tree leaves are *Atomic* A*bstract Queries* (defined in section 6.6) whereas they are triple patterns in SPARQL.

> *Note*. As an alternative, we could have used a relational algebra-based notation. However, extending it to account for the semantics of abstract atomic queries would have made the notation cumbersome and error prone. Thus, we felt like a notation based on usual SPARQL/SQL operator names was easier to manipulate and understand, while keeping the required expressive power.

The first INNER JOIN operator as well as the LEFT OUTER JOIN operator are entailed by the join constraints implied by shared variables. The second INNER JOIN notation, including the "AS child", "AS parent" and "ON child/<Ref> = parent/<Ref>" notations, is entailed by the join constraints expressed in xR2RML mappings using referencing object maps. Notation $\{v_1, \ldots v_n\}$, in the join operators, stands for the set of SPARQL variables on which the join is to be performed. "<Ref>" stands for any valid xR2RML data element reference, *i.e.* this shall be a column name for a tabular data source, an XPath expression for an XML database, a JSONPath expression for a NoSQL document store such as MongoDB and CouchDB, etc.

Definition 3. **Grammar of the Abstract Query Language**

```
<AbstractQuery> ::= <AtomicQuery> | <Query> |
                    <Query> FILTER <SPARQL filter> | <Query> LIMIT <integer>
<Query>         ::= <AbstractQuery> INNER JOIN <AbstractQuery> ON {v1, … vn} |
                    <AtomicQuery> AS child INNER JOIN <AtomicQuery> AS parent
                                  ON child/<Ref> = parent/<Ref> |
                    <AbstractQuery> LEFT OUTER JOIN <AbstractQuery> ON {v1, … vn} |
                    <AbstractQuery> UNION <AbstractQuery>
<AtomicQuery>   ::= {From, Project, Where, Limit}
<Ref>           ::= a valid xR2RML data element reference
```

The computation of the abstract operators shall be delegated to the target database if it supports them, *i.e.* if the target query language has equivalent operators (this is typically the case with a relational or an XML database), or they may be computed by the xR2RML query processing engine otherwise (this shall be the case of MongoDB, as we will show in Chapter 7).

6.4 Rewriting a SPARQL Graph Pattern into the Abstract Query Language

Function $trans_m$ (Definition 4) translates a well-designed SPARQL graph pattern [Pérez et al., 2009] into an abstract query, while making no assumption with respect to the target database query capabilities. It extends the translation algorithms defined in [Chebotko et al., 2009], [Unbehauen et al., 2013a] and [Priyatna et al., 2014].

***Definition 4**. Translation of a SPARQL graph pattern into an abstract query (function $trans_m$)*

Let *m* be an xR2RML mapping graph consisting of a set of xR2RML triples maps. Let *gp* be a well-designed SPARQL graph pattern, *f* be a SPARQL filter and *l* an integer value representing a limit (maximum number of results).

We denote by $\boldsymbol{trans_m}(gp, f, l)$ is the translation, under the set of mappings *m*, of "*gp* FILTER *f*" into an abstract query that shall not return more than *l* results.

We denote by $\boldsymbol{trans_m}(gp)$ the translation, under the set of mappings *m*, of *gp* into an abstract query. It is calculated by $\boldsymbol{trans_m}(gp, f, l)$ where *f* is set to true and *l* is set to the infinite value: $\boldsymbol{trans_m}(gp) = \boldsymbol{trans_m}(gp, \text{true}, \infty)$

Function $\boldsymbol{trans_m}(gp, \text{f}, l)$ is defined recursively as follows:

- if *gp* consists of a single triple pattern *tp*, $\boldsymbol{trans_m}(gp, f, l) = \boldsymbol{transTP_m}(tp, \boldsymbol{sparqlCond}(tp, f), l)$
- if *gp* is (*P* LIMIT *l'*), $\boldsymbol{trans_m}(gp, f, l) = \boldsymbol{trans_m}(P, f, \min(l, l'))$
- if *gp* is (*P* FILTER *f'*), $\boldsymbol{trans_m}(gp, f, l) =$
  $\boldsymbol{trans_m}(P, f \,\&\&\, f', \infty)$ **FILTER** $\boldsymbol{sparqlCond}(P, f \,\&\&\, f')$ **LIMIT** *l*
- if *gp* is (P_1 AND P_2), $\boldsymbol{trans_m}(gp, f, l) =$
  $\boldsymbol{trans_m}(P_1, f, \infty)$ **INNER JOIN** $\boldsymbol{trans_m}(P_2, f, \infty)$ **ON** $\text{var}(P_1) \cap \text{var}(P_2)$ **LIMIT** *l*
- if *gp* is (P_1 OPTIONAL P_2), $\boldsymbol{trans_m}(gp, f, l) =$
  $\boldsymbol{trans_m}(P_1, f, \infty)$ **LEFT OUTER JOIN** $\boldsymbol{trans_m}(P_2, \text{f}, \infty)$ **ON** $\text{var}(P_1) \cap \text{var}(P_2)$ **LIMIT** *l*
- if *gp* is (P_1 UNION P_2), $\boldsymbol{trans_m}(gp, f, l) =$
  $\boldsymbol{trans_m}(P_1, f, l)$ **UNION** $\boldsymbol{trans_m}(P_2, f, l)$ **LIMIT** *l*

where $\boldsymbol{transTP_m}$ is the translation of a single triple pattern into an abstract query (section 6.6) and ***sparqlCond*** discriminates SPARQL filter conditions (section 6.4.1).

Simplification: notations "FILTER true" and "LIMIT ∞" may be omitted.

Note. As we describe in subsequent sections 6.4.1 and 6.4.2, we deal with the SPARQL FILTER clause and LIMIT solution modifier in a way that pushes them down into the translation of each triple pattern, in order to make inner queries as selective as possible. For the sake of clarity, we do not consider SPARQL solution modifiers OFFSET, ORDER BY and DISTINCT. However, they could be managed in the very same way, *i.e.* as additional parameters of the $trans_m$ and $transTP_m$ functions and operators of the abstract query language.

Running Example. Let us give a first simple illustration: our running example does not include any SPARQL filter to keep it easy to follow. The application of the $trans_m$ function to the basic graph pattern (bgp) of the SPARQL query in Listing 15 is as follows:

$\text{trans}_m(\text{bgp})$
$= \text{trans}_m(\text{bgp}, \text{true}, \infty)$
$= \text{trans}_m(tp_1, \text{true}, \infty)$ **INNER JOIN** $\text{trans}_m(tp_2, \text{true}, \infty)$ **ON** $\text{var}(tp_1) \cap \text{var}(tp_2)$
$= \text{transTP}_m(tp_1, \text{true}, \infty)$ **INNER JOIN** $\text{transTP}_m(tp_2, \text{true}, \infty)$ **ON** {?dept}

6.4.1 Management of SPARQL filters

In the usual bottom-up evaluation of a SPARQL query, filters in the outer query do not contribute to the selectivity of inner-queries. A usual consequence, in SPARQL-to-SQL translations, is to rewrite a SPARQL FILTER into a SELECT-WHERE clause that encapsulates sub-queries. The problem in such a strategy is that sub-queries may return very large intermediate results. Consequently, this step is generally followed by an optimization phase, either by implementing specific SQL query optimizations or by relying on the underlying database engine.

In our generalized context, we do not know anything about the target database. Hence, we cannot assume (i) that the target query can be optimized, or (ii) that the database query evaluation engine is capable of such an optimization. We must therefore consider SPARQL filters at the earliest stage: we propose a generalized management of SPARQL filters that pushes down SPARQL filters into the translation of each triple pattern, in order to make inner queries as selective as possible, thereby limiting the size of intermediate results. This is achieved in function $trans_m$ by the introduction of a SPARQL filter argument initialized to "true" in the expression $trans_m(gp) = trans_m(gp, true, \infty)$. The filter argument shall be updated if the query graph pattern contains a FILTER clause.

> *Note.* Function $trans_m$ delegates the translation of each triple pattern into a sub-query to function $transTP_m$ (section 6.6). At this stage though, we do not yet explicit the way function $transTP_m$ deals with SPARQL filters. We just take care of the fact that, to filter data as early as possible, $transTP_m$ will need to know about the filters that are relevant for the translation of a given triple pattern. Therefore, we have to devise a method to select appropriate conditions from a SPARQL filter.

A SPARQL filter f can be considered as the conjunction of n conditions (n≥1): C_1 && ... C_n. The *sparqlCond* function is used to discriminate between these conditions with regards to two criteria:

(i) A condition C_i is pushed into the translation of triple pattern *tp* if all variables of C_i show in *tp*, *e.g.* a SPARQL condition involving variables ?x and ?y can be pushed into the translation of a triple pattern *tp* only if *tp* contains at least variables ?x and ?y.

(ii) A condition C_i is part of the abstract FILTER operator if at least one variable of C_i is shared by several triple patterns. This FILTER operator represents the join criteria. Example: if C_i contains variable ?x, and variable ?x shows in triple patterns tp_1 and tp_2, then condition C_i will be in the condition of the abstract FITLER operator.

Those two criteria are formalized in Definition 5 that defines how function *sparqlCond* discriminates between the filter conditions. Notice that the two criteria are not exclusive: a condition may match both criteria, and thereby show simultaneously in the FILTER operator and in the translation of a triple pattern.

Definition 5. Discrimination of SPARQL filter conditions (function sparqlCond)

Let *gp* be a well-designed SPARQL graph pattern, and TP_{gp} the set of triple patterns of *gp*.

Let *f* be the conjunctive SPARQL filter "C_1 && … && C_n", where C_1 to C_n are SPARQL conditions. Let $V(C_i)$ be the set of SPARQL variables named in condition C_i, and $V(tp)$ the set of SPARQL variables named in triple pattern *tp*.

Function ***sparqlCond*** is defined as follows:

- if *gp* consists of a single triple pattern *tp*, ***sparqlCond***(*tp*, *f*) is the conjunction of "true" and the conditions C_i such that $V(C_i) \subset V(tp)$.
- if *gp* is any other graph pattern, ***sparqlCond***(*gp*, *f*) is the conjunction of "true" and the conditions C_i such that $\exists v \in V(C_i), \exists tp_j, tp_k \in TP_{gp}, v \in V(tp_j) \cap V(tp_k)$.

Example. We illustrate this process with a dedicated example. We apply the $trans_m$ function to the SPARQL query Q depicted below, in which we denote by tp_1 to tp_3 the triple patterns and C_1 to C_2 the conditions of the conjunctive SPARQL filter.

```
SELECT ?x WHERE
{ ?x foaf:mbox ?mbox1.                          // tp1
  ?y foaf:mbox "john@foo.com".                  // tp2
  ?x foaf:knows ?y.                             // tp3
  FILTER {
     contains(str(?mbox1), "foo.com") &&        // C1
     ?x != ?y }                                 // C2
}
```

Let us compute function *sparqlCond* for each triple pattern:

- tp_1 involves variables `?x` and `?mbox1`. No condition involves both variables, but C_1 involves `?mbox1` and no other variable, thus C_1 matches criteria (i) above. C_2 involves `?x` but it also involves `?y` that is not in tp_1. Hence:

  sparqlCond(tp_1, C_1 && C_2) = C_1

- tp_2 has one variable, `?y`, and no condition involves only `?y`. Hence no condition can be pushed into the translation of tp_2:

  sparqlCond(tp_2, C_1 && C_2) = true

- tp_3 has two variables `?x` and `?y`, and only condition C_2 involves both of them. Hence:

  sparqlCond(tp3, C_1 && C_2) = C_2

- Lastly, only condition C_2 involves variables shared by several triple patterns: `?x` and `?y`. This entails a clause "FILTER C_2".

Finally, by applying function $trans_m$ to Q, we come up with the following abstract query:

```
transm(Q) =
     transTPm(tp1, C1, ∞) INNER JOIN transTPm(tp2, true, ∞) ON ∅
                          INNER JOIN transTPm(tp3, C2, ∞) ON {?x,?y}
     FILTER C2
```

> Notice that condition C_2 is used twice: firstly, in the translation of tp_3 because all variables of C_2 are in tp_3; secondly, in the FILTER clause as it involves variables ?x and ?y which are shared by two triple patterns.

6.4.2 Management of the LIMIT clause

The way we deal with the LIMIT solution modifier is motivated by the same concern as in the case of SPARQL filters. In the bottom-up evaluation of a SPARQL query, the LIMIT in the outer query does not contribute to the selectivity of inner-queries. Thus, sub-queries may return unnecessary large intermediate results. This issue is generally taken care of by implementing specific query optimizations or by relying on the underlying database engine to do the optimization. But again, in our generalized context, we do not know whether (i) it will be possible to optimize the target query, nor (ii) whether the database query evaluation engine is capable of such an optimization.

Therefore, we propose a method to push down the LIMIT solution modifier into the translation of each triple pattern, in order to make inner queries as selective as possible. Function $trans_m$ has a limit argument *l*, initialized to "∞" in the expression $\texttt{trans}_m\texttt{(gp) = trans}_m\texttt{(gp, true, } \infty\texttt{)}$. The limit argument shall be modified depending on the type of graph pattern passed to $trans_m$. Below we elaborate on the different situations encountered in Definition 4:

- In the rule:

  $\boldsymbol{trans_m}(P \text{ LIMIT } l', f, l) = \boldsymbol{trans_m}(P, f, min(l, l'))$

  the SPARQL LIMIT *l'* is passed to the subsequent graph pattern translation. If there is already a limit (from the outer query), then the smallest limit is considered, hence the parameter *min(l, l')*.

- In the case of a simple triple pattern, the limit argument is passed to the $transTP_m$ function:

  $\boldsymbol{trans_m}(tp, f, l) = \boldsymbol{transTP_m}(tp, \boldsymbol{sparqlCond}(tp, f), l)$

- In a graph pattern *P FILTER f'*, we cannot know in advance how many results will be filtered out by the FILTER clause. Consequently, we have to run the query with no limit and apply the filter afterwards. This explains the "∞" parameter in:

  $\boldsymbol{trans_m}(gp, f, l) = \boldsymbol{trans_m}(P, f \,\&\&\, f', \infty)$ **FILTER** $\boldsymbol{sparqlCond}(P, f \,\&\&\, f')$ **LIMIT** *l*

- Similarly, in the case of an inner or left join, we cannot know in advance how many results will be returned. Consequently, we have to run both queries with no limit, apply the join, and only then limit the number of results. Hence the "∞" parameter in expressions:

  $\boldsymbol{trans_m}(P_1, f, \infty)$ **INNER JOIN** $\boldsymbol{trans_m}(P_2, f, \infty)$ **ON** $\text{var}(P_1) \cap \text{var}(P_2)$ **LIMIT** *l*

  and

  $\boldsymbol{trans_m}(P_1, f, \infty)$ **LEFT OUTER JOIN** $\boldsymbol{trans_m}(P_2, f, \infty)$ **ON** $\text{var}(P_1) \cap \text{var}(P_2)$ **LIMIT** *l*

- Finally, in the union case, none of the two operands of the UNION should return more than the limit parameter. Hence the parameter *l* in each operand. Besides, the whole UNION should return more than *l* either, hence the LIMIT solution modifier.

  $\boldsymbol{trans_m}(P_1 \text{ UNION } P_2, f, l) = \boldsymbol{trans_m}(P_1, f, l)$ **UNION** $\boldsymbol{trans_m}(P_2, f, l)$ **LIMIT** *l*

6.5 Binding xR2RML Triples Maps to Triple Patterns

Before defining function $transTP_m$ that translates SPARQL triple patterns into atomic abstract queries, in this section we elaborate on how to figure out which xR2RML triples maps are likely to generate RDF triples matching the SPARQL triple patterns.

In the following, we assume that xR2RML triples are normalized in the sense defined in section 6.1.1. We denote by *TM.sub*, *TM.pred* and *TM.obj* respectively the subject map, the predicate map and the object map of the normalized triples map *TM*. Furthermore, in Definition 6 we adapt the concept of *triple pattern binding* introduced by Unbehauen et al [Unbehauen et al., 2013a].

> ***Definition 6. Triple Pattern Binding***
>
> Let *m* be an xR2RML mapping graph consisting of a set of xR2RML triples map, and *tp* be a triple pattern.
>
> A triples map $TM \in m$ is **bound to *tp*** if it is likely to produce triples matching *tp*.
>
> A **triple pattern binding** is a pair (*tp*, *TMSet*) where *TMSet* is the set of triples maps of *m* that are bound to *tp*.

Function $bind_m$ (Definition 7) determines, for a graph pattern *gp*, the bindings of each triple pattern of *gp*. It takes into account join constraints implied by shared variables and by cross-references defined in the mapping (xR2RML referencing-object map), and the SPARQL filter constraints whose unsatisfiability can be verified statically. This is achieved by means of two functions: *compatible* and *reduce*. These functions were introduced by Unbehauen et al. in the SPARQL-to-SQL context [Unbehauen et al., 2013a], but important details were left untold. Especially, the authors did not formally define what the compatibility between a term map and a triple pattern term means, and more importantly they did not investigate the static compatibility between a term map and a SPARQL filter. In the subsequent sections, we define these functions in details and extend them to fit in the context of our abstract query language.

> ***Running Example***: Before we get into the details, let us illustrate informally the way function $bind_m$ infers triple pattern bindings. Triple pattern tp_1 is as follows:
>
> `<http://example.org/staff/Dunbar> ex:manages ?dept.`
>
> The subject term, `<http://example.org/staff/Dunbar>`, could be produced by the template string in the subject map of triples map <#Staff>:
>
> `"http://example.org/staff/{$['lastname','familyname']}"`
>
> On the contrary, it could not be produced by the template string in triples map <#Department>:
>
> `"http://example.org/dept/{$.code}"`
>
> Additionally, the predicate part of tp_1, `ex:manages`, matches the constant predicate map of triples map <#Staff>. Consequently, triples map <#Staff> may generate triples that match tp_1; we say that triples map <#Staff> is bound to tp_1. Conversely, <#Department> can certainly not generate triples that match tp_1.

The very same reasoning allows to deduce that triple pattern tp_2 may be produced by triples map <#Department>, but not by triples map <#Staff>. Finally, we obtain the following bindings:

$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}), (tp_2, \{<\#Departments>\}) \}$

Definition 7. Binding of xR2RML mappings to SPARQL triple pattern (function $bind_m$)

Let *m* be an xR2RML mapping graph consisting of a set of xR2RML triples maps, *gp* be a well-designed graph pattern, and *f* be a SPARQL filter.

We denote by ***$bind_m$(gp, f)*** the set of triple pattern bindings of "*gp* FILTER *f*" under *m*.

We denote by ***$bind_m$(gp)*** the set of triple pattern bindings of *gp* under *m*. It is calculated by ***$bind_m$***(*gp*, *f*) where *f* is set to true: ***$bind_m$***(*gp*) = ***$bind_m$***(*gp*, true)

Function ***$bind_m$***(*gp*, *f*) is defined recursively as follows:

- if *gp* consists of a single triple pattern *tp*,
  - if *tp.pred* is one of `rdf:first`, `rdf:rest`, `rdf:nil`, `rdf:_1`, `rdf:_2` etc., or *tp.pred* is `rdf:type` and *tp.obj* is one of `rdf:List`, `rdf:Bag`, `rdf:Seq`, `rdf:Alt`, then *tp* is ignored.
  - Otherwise, ***$bind_m$***(*gp*, *f*) is the pair (*tp*, *TMSet*) where *TMSet* = {*TM* | *TM* ∈ *m* ∧ ***compatible***(*TM.sub*, *tp.sub*, *f*) ∧ ***compatible***(*TM.pred*, *tp.pred*, *f*) ∧ ***compatible***(*TM.obj*, *tp.obj*, *f*)}
- if *gp* is (P_1 AND P_2), ***$bind_m$***(*gp*, *f*) = ***reduce***(***$bind_m$***(P_1, *f*), ***$bind_m$***(P_2, *f*)) ∪ ***reduce***(***$bind_m$***(P_2, *f*), ***$bind_m$***(P_1, *f*))
- if *gp* is (P_1 OPTIONAL P_2), ***$bind_m$***(*gp*, f) = ***$bind_m$***(P_1, f) ∪ ***reduce***(***$bind_m$***(P_2, *f*), ***$bind_m$***(P_1, *f*))
- if *gp* is (P_1 UNION P_2), ***$bind_m$***(*gp*, f) = ***$bind_m$***(P_1, f) ∪ ***$bind_m$***(P_2, *f*)
- if *gp* is (*P* FILTER *f'*), ***$bind_m$***(*gp*, f) = ***$bind_m$***(*P*, *f* && *f'*)

where ***compatible*** verifies the compatibility between a term map, a triple pattern term and a SPARQL filter (section 6.5.2), and ***reduce*** utilizes dependencies between graph patterns to reduce their bindings (section 6.5.3).

6.5.1 Case of RDF Collections and Containers

An xR2RML term map may generate RDF collections and containers using specific term type values `xrr:RdfList`, `xrr:RdfSeq`, `xrr:RdfBag` and `xrr:RdfAlt`. Some SPARQL queries may refer to the members of such collections and containers. In our running example, let us assume that triples map `<#Departments>` generates an RDF list of senior members instead of one triple per senior member, with the amended predicate-object map below:

```
rr:predicateObjectMap [
   rr:predicate ex:seniorMembers;
   rr:objectMap [
      xrr:reference "$.members[?(@.age >= 40)].name";
      rr:termType xrr:RdfList. ] ]. # added to generate an rdf:List
```

In this context, a user may issue a SPARQL query about the senior members, such as:

```
SELECT ?first WHERE {
    ?dept ex:seniorMembers ?seniors.    // tp1
    ?seniors rdf:type  rdf:List.        // tp2
    ?seniors rdf:first ?first.          // tp3
}
```

Trivially, triples map `<#Departments>` can be bound to triple pattern tp_1 since their predicate parts match (`ex:seniorMembers`). Let us now consider tp_2 and tp_3. In our running example, there is no triples map with constant predicate "`rdf:type`" and constant object "`rdf:List`" that could be bound to tp_2. Similarly, there is no triples map with constant predicate "`rdf:first`" that could be bound to tp_3. With no bindings, no RDF triples matching tp_2 nor tp_3 are generated. Consequently, the join (logical AND) between triples patterns tp_1, tp_2 and tp_3 will return an empty result set, whereas there are actually solutions to that SPARQL query.

To avoid coming up with empty bindings, the $bind_m$ function ignores triple patterns that pertain to RDF collections and containers (triple patterns whose predicate is one of `rdf:first`, `rdf:rest`, `rdf:nil`, `rdf:_1`, `rdf:_2` etc., or the predicate is `rdf:type` and the object is one of `rdf:List`, `rdf:Bag`, `rdf:Seq`, `rdf:Alt`). This does not prevent from generating a query, executing it and obtaining relevant results; what we do is actually to simplify the SPARQL query by ignoring those triple patterns that we cannot deal with at this stage, thus making the SPARQL query less specific. As a result, the generated target database query may return more results than actually expected; finally, we may generate RDF triples that do not match the SPARQL query. To work out this issue, we propose to perform a *late SPARQL query evaluation* that shall rule out all unneeded triples. This step is described further on in section 7.5.

6.5.2 Compatibility of Term Maps, Triple Pattern Terms and SPARQL Filters

To decide whether a triples map can be bound to a triple pattern, we must verify whether the triples map can potentially generate RDF triples matching the triple pattern. More precisely, we look for incompatibilities between each term map of the triples map, and the corresponding term in the triple pattern (a triple pattern term may be a literal, an IRI, a blank node or a variable). In Definition 6, this is denoted by the expression:

$$compatible(TM.sub, tp.sub, f) \wedge compatible(TM.pred, tp.pred, f) \wedge compatible(TM.obj, tp.obj, f)$$

Function *compatible* (Definition 8) checks if a term map (*termMap*) is compatible with a term of a triple pattern (*tpTerm*) and a SPARQL filter *f*, *i.e.* it verifies whether there is any contradiction between *termMap* and *tpTerm*, or between *termMap* and *f*. Unbehauen et al. defined the compatibility of *termMap* and *tpTerm* as: $tpTerm \in range(termMap)$, but no description of the *range* function was provided. We precise this definition below.

A term map is always considered compatible with a variable, unless a SPARQL filter contradicts the term map. The later situation is identified in function *compatibleFilter* (Definition 9). It pertains to type constraints expressed using SPARQL operators *isIRI*, *isLiteral* or *isBlank*, as well as language and data type constraints expressed using operators *lang, langMatches* or *datatype*. For instance, if variable

`?var` is matched with an object map that produces literals (`rr:termType rr:Literal`), and the SPARQL filter contains a necessary condition `isIRI(?var)`, then this condition is unsatisfiable.

When the triple pattern term is not a variable, function *compatible* identifies the similar situations wherein the triple pattern term and the term map cannot match with regards to the type of the triple pattern term[76] (literal, IRI, blank node), its language tag (e.g. `"string"@en`) or its data type (e.g. `10^^xsd:integer`).

Definition 8. Compatibility between a term map, a triple pattern term and a SPARQL filter (function compatible)

Let *tpTerm* be a term of a triple pattern, *termMap* be a term map of an xR2RML triples map *TM* and *f* be a SPARQL filter.

termMap is compatible with *tpTerm* and *f*, denoted by **compatible**(*termMap*, *tpTerm*, *f*), iif *termMap* is compatible with filter *f* (see Definition 9) and either (i) *tpTerm* is a variable or (ii) none of the following assertions holds:

- *tpTerm* is a literal and the term type of *termMap* is not `rr:Literal`;
- *tpTerm* is an IRI and the term type of *termMap* is not `rr:IRI`;
- *tpTerm* is a blank node and the term type of *termMap* is none of {`rr:BlankNode`, `xrr:RdfList`, `xrr:RdfBag`, `xrr:RdfSeq`, `xrr:RdfAlt`}[77];
- *tpTerm* is a literal with a language tag *L*, and the language of *termMap* is either undefined or different from *L*;
- *tpTerm* is a literal with a datatype *T*, and the datatype of *termMap* is either undefined or different from *T*;
- *termMap* is constant-valued with value *V*, and *tpTerm* is different from *V*;
- *termMap* is template-valued with template string *T*, and *tpTerm* does not match *T*;
- *termMap* is a ReferencingObjectMap and the subject map of the parent triples map is not compatible with *tpTerm*, i.e. **¬compatible**(*termMap*.parentTriplesMap.subjectMap, *tpTerm*, *f*).

[76] Let us remind that the term type may be explicitly stated with the `rr:termType` property, or have a default value. For instance, a template-valued term map has the rr:IRI default term type and a reference-valued term map has the rr:Literal default term type.

[77] As per the xR2RML specification, term types `xrr:RdfList`, `xrr:RdfBag`, `xrr:RdfSeq`, `xrr:RdfAlt` yield RDF collections and containers implemented as blank nodes.

***Definition 9. Compatibility between a term map and a SPARQL filter (function compatibleFilter*)**

Let *termMap* be an xR2RML term map and *f* be a SPARQL filter.

termMap is compatible with *f*, denoted by **compatibleFilter**(*termMap*, *f*), iff either *f*="true" or none of the following assertions holds:

- a necessary condition of *f* is `isIRI(?var)` and the term type of *termMap* is not `rr:IRI`;
- a necessary condition of *f* is `isLiteral(?var)` and the term type of *termMap* is not `rr:Literal`;
- a necessary condition of *f* is `isBlank(?var)` and the term type of *termMap* is none of {`rr:BlankNode`, `xrr:RdfList`, `xrr:RdfBag`, `xrr:RdfSeq`, `xrr:RdfAlt`};
- a necessary condition of *f* is `lang(?var)="L"` or `langMatches(lang(?var),"L")`, and the language of *termMap* is either not defined or different from L;
- a necessary condition of *f* is `datatype(?var)=<T>` and the datatype of *termMap* is either undefined or different from <T>.

6.5.3 Reduction of Bindings

At this point, we have come up with bindings of xR2RML triples maps to each triple pattern of the SPARQL graph pattern. Bindings are computed for each triple pattern independently of the others. Yet, triple patterns are not independent of each other: shared variables induce join constraints that can help us find out inconsistent bindings. For instance, consider two triple patterns tp_1 and tp_2 that have a shared variable *?v*, triples map TM_1 is bound to tp_1 and triples map TM_2 is bound to tp_2. If the term map associated to *?v* in TM_1 generates literals, whereas the term map associated to *?v* in TM_2 generates IRIs, these bindings are incompatible. Consequently, we can rule out TM_1 from the bindings of tp_1 and TM_2 from the bindings of tp_2. This is what we call "reduction of bindings".

First, we define the concept of compatibility between two term maps tm_1 and tm_2. Unbehauen et al. define it as the condition: $range(tm_1) \cap range(tm_2) \neq \emptyset$, but no description of the *range* function is provided, which leaves room for interpretation. In Definition 10, we give a formal description of what it means in our context.

Definition 10. Compatibility between two term maps (function compatibleTermMaps)

Let tm_1 and tm_2 be two xR2RML term maps. tm_1 and tm_2 are compatible, denoted by **compatibleTermMaps**(tm_1, tm_2), if none of the following assertions holds:

(1) tm_1 and tm_2 have different term types (`rr:Literal`, `rr:BlankNode`, `rr:IRI`, `xrr:RdfList`, `xrr:RdfSeq`, `xrr:RdfBag`, `xrr:RdfAlt`).

(2) tm_1 and tm_2 have different language tags, or one has a language tag and the other does not.

(3) tm_1 and tm_2 have different data types, or one has a data type and the other does not.

(4) tm_1 and tm_2 are both template-valued, and they have incompatible template strings.

(5) tm_1 (respectively tm_2) is a ReferencingObjectMap and the subject map of its parent triples map is not compatible with tm_2 (resp. tm_1), *i.e.* **¬compatibleTermMaps**(tm_1.parentTriplesMap.subjectMap, tm_2), (respectively **¬compatibleTermMaps**(tm_1, tm_2.parentTriplesMap.subjectMap)).

Note. Any of the assertions (1) to (5) is a sufficient condition to make two term maps incompatible. The difference between term maps types (constant-valued, reference-valued or template-valued) cannot be used as an incompatibility criterion. For instance, a reference-valued term map returning a URL from the database with a term type `rr:IRI` could be compatible with a template-valued term map building a URL from some other value of the database.

Assertion (4) involves the concept of incompatibility between template strings. Informally, two template strings are incompatible if their fixed parts are incompatible. *E.g.* "`http://example.org/{`*xx*`}/B{`*yy*`}`", "`http://example.org/{`*zz*`}`" and "`{`*tt*`}`" are pairwise compatible, but "`http://example.org/{`*xx*`}`" and "`http://example.com/{`*yy*`}`" are not compatible.

Now, we define function *join* that examines the variables shared by two triple patterns to detect unsatisfiable join constraints:

Definition 11. Detect unsatisfiable bindings (function join)

Let m be an xR2RML mapping graph consisting of a set of xR2RML triples map.

Let tp_1 and tp_2 be triple patterns, $V = var(tp_1) \cap var(tp_2)$ be the set of variables shared by tp_1 and tp_2, and $tpb_1=(tp_1, TMSet_1)$ and $tpb_2=(tp_2, TMSet_2)$ be the triple pattern bindings of tp_1 and tp_2, with $TMSet_1 \subseteq m$ and $TMSet_2 \subseteq m$.

Let pos_{tp}: $V \rightarrow$ {sub, pred, obj} be the function that returns the position of a variable $v \in V$ in triple pattern tp.

We denote by **join**(tpb_1, tpb_2) the set of pairs of triples maps (TM_1, TM_2) $\in TMSet_1 \times TMSet_2$, such that, for each $v \in$ V, it holds that **compatibleTermMaps**($TM_1.pos_{tp1}(v)$, $TM_2.pos_{tp2}(v)$).

If V is empty (*i.e.* tp_1 and tp_2 have no common variable), **join**(tpb_1, tpb_2) is the set of all pairs (TM_1, TM_2) $\in TMSet_1 \times TMSet_2$.

In other words, function *join* returns the pair (TM_1, TM_2) if, for each variable v shared by tp_1 and tp_2, the term map associated to v in TM_1 is compatible with the term map associated to v in TM_2.

Example. Let us consider two triple patterns tp_1 *and* tp_2 with a shared variable ?y:

tp_1 = `?x foaf:knows ?y` ⇨ pos_{tp1}(?y) = *obj*.

tp_2 = `?y foaf:knows <#me>` ⇨ pos_{tp2}(?y) = *sub*.

We assume the following bindings: $tpb_1 = (tp_1, \{TM_{1a}, TM_{1b}\})$, $tpb_2 = (tp_2, \{TM_{2a}, TM_{2b}\})$. Finally, we assume the following compatibility matrix between the term maps of the triples maps bound to tp_1 and tp_2:

	TM_{1a}.obj	TM_{1b}.obj
TM_{2a}.sub	✓	✓
TM_{2b}.sub	✗	✗

In this context, we obtain the following pairs of triples maps:

$join(tpb_1, tpb_2) = \{ (TM_{1a}, TM_{2a}), (TM_{1b}, TM_{2a}) \}$

$join(tpb_2, tpb_1) = \{ (TM_{2a}, TM_{1a}), (TM_{2a}, TM_{1b}) \}$

We can now define function *reduce* (Definition 12), that computes the minimal set of triples maps bound to each triple pattern (minimal with respect to the join constraints implied by shared variables).

Definition 12. Reduction of bindings (function reduce)

Let *m* be an xR2RML mapping graph consisting of a set of xR2RML triples map, $tpb_1 = (tp_1, TMSet_1)$ and $tpb_2 = (tp_2, TMSet_2)$ be triple pattern bindings with $TMSet_1 \subseteq m$ and $TMSet_2 \subseteq m$.

We denote by **reduce**(tpb_1, tpb_2) the reduced bindings of tp_1, *i.e.* the reduction of the bindings of tp_1 with the bindings of tp_2, defined as the set of triples maps that appear as the left component of the pairs in **join**(tpb_1, tpb_2). Formally:

reduce(tpb_1, tpb_2) = (tp_1, $\{TM_i \in TMSet_1, \exists (TM_i, TM_j) \in join(tpb_1, tpb_2)\}$)

Furthermore, we denote by **reduce**(TPB_1, TPB_2) the reduction of the set of triple pattern bindings TPB_1 with the set of triple pattern bindings TPB_2, defined as the union of the pairwise reduction of the triple pattern bindings of TPB_1 and TPB_2. Formally:

reduce(TPB_1, TPB_2) = { (*tp*, *TMSet*), $\exists$ (tpb_1, tpb_2) $\in TPB_1 \times TPB_2$ such that

(*tp*, *TMSet*) = **reduce**(tpb_1, tpb_2) }

The latter definition is used to reduce the bindings, not only of triple pattern bindings, but also of graph patterns.

Example. Following up on the previous example, function *reduce* makes a simple projection of the left component of the pairs computed by function *join*:

join(tpb_1, tpb_2) = { (**TM_{1a},** TM_{2a}), (**TM_{1b},** TM_{2a}) }
reduce(tpb_1, tpb_2) = (tp_1, {**TM_{1a}, TM_{1b}**})

join(tpb_2, tpb_1) = { (**TM_{2a},** TM_{1a}), (**TM_{2a},** TM_{1b}) }
reduce(tpb_2, tpb_1) = (tp_2, {**TM_{2a}**})

6.6 Translation of a SPARQL Triple Pattern into Atomic Abstract Queries

The $trans_m$ function delegates to the $transTP_m$ function (Definition 13) the translation a single triple pattern *tp* into an abstract query under the reduced set of compatible xR2RML triples maps (the triples maps of *m* bound to *tp* by function $bind_m$). Below, we define function $transTP_m$, then the concept of *Atomic Abstract Query*. In subsequent sections, we go through the algorithm of $transTP_m$ and the details of how atomic abstract queries are computed.

Definition 13. Function $transTP_m$

Let *m* be an xR2RML mapping graph consisting of a set of xR2RML triples maps, *gp* be a well-designed graph pattern, and *tp* a triple pattern of *gp*. Let *l* be the maximum number of query results, and *f* be a SPARQL filter expression. Let $getBoundTMs_m$(*gp*, *tp*, *f*) be the function that, given *gp*, *tp* and *f*, returns the set of triples maps of *m* that are bound to *tp* in $bind_m$(*gp*, *f*).

We denote by **$transTP_m$**(*tp*, *f*, *l*) the translation, under $getBoundTMs_m$(*gp*, *tp*, *f*), of *tp* into an abstract query whereof results can be translated into at most *l* RDF triples matching "*tp* FILTER *f*". The resulting abstract query, denoted `<ResultQuery>` in the grammar below, is a union of per-triples-map subqueries, where a subquery is either an *Atomic Abstract Query* or the inner join of two *Atomic Abstract Queries* (see details in Algorithm 1).

```
<ResultQuery> ::= <SubQuery> (UNION <SubQuery>)*
<SubQuery>    ::= <AtomicQuery> |
                  <AtomicQuery> AS child INNER JOIN <AtomicQuery> AS parent
                                ON child/<Ref> = parent/<Ref>
```

Definition 14. Atomic Abstract Query

An *Atomic Abstract Query* is an abstract query obtained by matching a SPARQL triple pattern *tp* with an xR2RML triples map bound to *tp*. It is denoted by **{From, Project, Where, Limit}**, where:

- "**From**" consists of the triples map's logical source;
- "**Project**" is the set of xR2RML data element references that are projected, i.e. returned as part of the query results. There are three types of projection:
  - <*xR2RML reference*>
  - <*xR2RML reference*> **AS** <*SPARQL variable*>
  - <*Constant value*> **AS** <*SPARQL variable*>
- "**Where**" is a conjunction of conditions on xR2RML data element references, entailed by matching the terms of *tp* with (i) their corresponding term map in the triples map, or (ii) with a SPARQL filter. Three types of condition exist:
  - **isNotNull**(<*xR2RML reference*>)
  - **equals**(value, <*xR2RML reference*>)
  - **sparqlFilter**(<*SPARQL filter*>)
- **"Limit"** is the maximum number of results that must be returned by the atomic query.

The abbreviated notation **{From, Project, Where}** may be used when Limit is ∞.

6.6.1 Algorithm of Function $transTP_m$

Function $transTP_m$ translates a triple pattern into an abstract query under the reduced set of triples maps bound to it. Algorithm 1 describes function $transTP_m$ in further details. It consists of a loop on all triples maps bound to *tp* (line 4 to 24). The result query is a UNION of all per-triples-map subqueries (line 23). For each triples map *TM*, the algorithm constructs the *From*, *Project* and *Where* parts of an atomic abstract query (lines 5-7). Then, two cases are distinguished:

- When the object map is a regular object map (no cross-reference), a single atomic abstract query is created (line 21): {*From, Project, Where*}.
- When the object map is a referencing object map, *e.g.* child triples map TM_1 produces the subject and predicate terms while parent triples map TM_2 produces object terms, a second atomic abstract query is constructed (lines 12-14) to account for TM_2: *PFrom* is TM_2's logical source, *PProject* projects the xR2RML data element references of TM_2's subject map, and *PWhere* embeds conditions on the xR2RML data element references of TM_2's subject map. Then, the abstract query corresponding to the couple (*tp*, TM_1) is the INNER JOIN of the two atomic abstract queries (lines 15-18):

```
{From, Project, Where, ∞} AS child
INNER JOIN
{PFrom, PProject, PWhere, ∞} AS parent
ON child/childRef = parent/parentRef
```

  where `childRef` and `parentRef` denote the values of properties `rr:child` and `rr:parent` respectively.

Note: Interestingly, we observe that the abstract INNER JOIN operator may be entailed by the conjunction of SPARQL basic graph patterns (as shown in function $trans_m$), but also by cross-

references denoted in the mappings (as exemplified above). Similarly, the abstract UNION operator may arise from the translation of the SPARQL UNION clause or from the binding of several triples maps to the same triple pattern (as explained above).

If the algorithm entails an INNER JOIN or UNION operator, the limit parameter, l, is managed as described in section 6.4.2. It may be used as the *Limit* part of an atomic abstract query and with the *LIMIT* abstract query operator.

Algorithm 1: Translation of a triple pattern tp into an abstract query (function $transTP_m$).
f is a SPARQL filter, l is the maximum number of results.

```
1  Function transTP_m(tp, f, l):
2    Query ← <empty query>
3    BoundTMs ← getBoundTMs_m(gp, tp, f)
4    for each TM ∈ BoundTMs do
5      From ← <TM's logicalSource>
6      Project ← genProjection(tp, TM)      # (see Appendix C)
7      Where ← genCond(tp, TM, f)           # (see Appendix C)
8      OM ← TM.predicateObjectMap.objectMap
9      if OM is a referencing object map then
10       childRef ← OM.joinCondition.child
11       parentRef ← OM.joinCondition.parent
12       PFrom ← <OM.parentTriplesMap's logical source>
13       PProject ← genProjectionParent(tp, TM)    # (see Appendix C)
14       PWhere ← genCondParent(tp, TM, f)         # (see Appendix C)
15       Q ←  {From, Project, Where, ∞} AS child
16            INNER JOIN
17            {PFrom, PProject, PWhere, ∞} AS parent
18            ON child/childRef = parent/parentRef
19            LIMIT l
20     else
21       Q ← {From, Project, Where, l}
22     end if
23     Query ← Query UNION Q LIMIT l
24   end for
25   return Query
```

Running Example. We have already shown that:

$bind_m(bgp) = \{ (tp_1, \{<\#Staff>\}), (tp_2, \{<\#Departments>\}) \}$

Hence,

$getBoundTMs_m(gp, tp_1, true) = \{<\#Staff>\}$
$getBoundTMs_m(gp, tp_2, true) = \{<\#Departments>\}$

Now, let us run function *transTP*$_m$ for tp_2:

```
tp2 = ?dept ex:hasSeniorMember ?senior.
transTPm(tp2, true, ∞) =
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← genProjection(tp2, <#Departments>),
      Where   ← genCond(tp2, <#Departments>, true)
      Limit   ← ∞ }
```

In the case of tp_1, the bound triples map, <#Staff>, contains a referencing object map. Consequently, the translation entails an inner join on the xR2RML references mentioned in the *rr:joinCondition* property of the referencing object map:

```
tp1 = <http://example.org/staff/Dunbar> ex:manages ?dept
transTPm(tp1, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← genProjection(tp1, <#Staff>)
      Where   ← genCond(tp1, <#Staff>, true)
      Limit   ← ∞
    } AS child
    INNER JOIN
    { PFrom    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      PProject ← genProjectionParent(tp1, <#Staff>)
      PWhere   ← genCondParent(tp1, <#Staff>, true)
      Limit    ← ∞
    } AS parent
    ON child/$.manages.* = parent/$.dept
    LIMIT ∞
```

Note: From now on, we shall omit the *Limit* part in an atomic query, and the *LIMIT* abstract query operator, when the limit value is ∞.

6.6.2 Computing Atomic Abstract Queries

We now go through further details about how the *From, Project* and *Where* parts of an abstract atomic query are computed. The detailed algorithms of functions *genProjection, genProjectionParent, genCond* and *genCondParent* are provided in Appendix C.

From. The *From* part provides the concrete query that the abstract query relies on. It contains the logical source of triples map TM that consists of the `xrr:query` or `rr:tableName` properties, an optional iterator (property `rml:iterator`) and the optional `xrr:uniqueRef` property. In the running example, the *From* part for tp_2 is simply:

```
{[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
```

In the case of tp_1, two atomic abstract queries are created, each one referring to the logical source of one triples map.

Project. The projection part of a database query restricts the set of attributes that must be returned in the query response. In relational algebra, this would be denoted by the operator π: $\pi_{a1, \ldots an}(R)$ is the set obtained when the components of the tuple R are restricted to the set $\{a_1, \ldots, a_n\}$. In the context of a relational database, the attributes are columns, whereas in the context of a JSON document store, attributes are fields of JSON documents.

The *genProjection* and *genProjectionParent* functions (appendix C.1) select the xR2RML data element references that must be projected. When an xR2RML reference is matched with a SPARQL variable in the triple pattern, it is projected with notation "AS <*variable name*>". In the running example, the subject and object of tp_2 are variables "?dept" and "?senior", respectively matched with subject map's reference "`$.code`" and object map's reference "`$.members[?(@.age >= 40)].name`". Consequently:

```
genProjection(tp2, <#Departments>) =
    { $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior }
```

How the JSON fields named in the JSONPath expressions (code, members, age, name) are actually projected is not relevant at this point: the atomic abstract query simply specifies the xR2RML references to be projected, it does not tell how they will be projected in a concrete target database query.

Additionally, when a referencing object map is involved (cross-reference), functions *genProjection* and *genProjectionParent* project the joined references mentioned in an xR2RML `rr:joinCondition` property. This is illustrated with tp_1 in the running example. Triples map <#Staff> has a referencing object map whose child and parent references are projected: function *genProjection* projects the child reference "`$.manages.*`", while function *genProjectionParent* projects the parent reference "`$.dept`":

```
genProjection(tp1, <#Staff>) = {$.manages.*}
genProjectionParent(tp1, <#Staff>) = {$.dept, $.code AS ?dept}
```

Note that, since the joined references are not matched with a variable of the SPARQL query, they are projected without the AS operator.

A last case concerns constant term maps: when a SPARQL variable is matched with a constant term map, that constant value is projected as the variable. For instance, let us consider a new triple pattern tp_3:

```
<http://example.org/staff/Dunbar> ?predicate ?dept
```

Variable `?predicate` is matched with the constant predicate map in triples map <#Staff>. To account for this constant projection, we would write:

```
genProjection(tp3, <#Staff>) = {ex:manages AS ?predicate}
```

Where. The *genCond* function (section C.2) computes the *Where* part by matching each triple pattern term with its corresponding term map. In relational algebra, this would be denoted by the selection operator σ: $\sigma_{\varphi}(R)$ selects all tuples in R for which the proposition φ holds. In our context, R is the triples map logical source, and φ is a conjunction of conditions of three types: non-null, equality or SPARQL filter. Below, we determine the type of condition entailed according to the type of triple pattern term and the type of term map that are matched with each other.

(a) A SPARQL variable in the triple pattern entails a non-null condition on the corresponding xR2RML reference(s). Let us exemplify this: the subject part of tp_2 is variable `?dept`; it is matched with the subject map of triples map <#Departments>, whose template string is "`http://example.org/dept/{$.code}`". Without any further knowledge on `?dept`, the match simply states that the subject map must return a valid value, in other words the reference "`$.code`" must not return "null". This entails a condition: `isNotNull($.code)`. When applied to the object of tp_2, the same method entails a second non-null condition: `isNotNull($.members[?(@.age >= 40)].name)`.

As a result, we can already deduce the evaluation of function *genCond* on tp_2:

```
genCond(tp2, <#Departments>, true) = { isNotNull($.code),
                                       isNotNull($.members[?(@.age >= 40)].name) }
```

(b) A constant term in the triple pattern (literal or IRI) entails an equality condition. In our running example, the subject part of tp_1, `<http://example.org/staff/Dunbar>`, is matched with the template string "`http://example.org/staff/{$['lastname','familyname']}`" of <#Staff>'s subject map. This entails the equality condition:

```
equals("Dunbar", $['lastname','familyname']),
```

stating that either "lastname" or "familyname" must equal "Dunbar".

(c) When a constant term map is matched with a triple pattern term,

- If the triple pattern term is also constant (literal or IRI), then no condition is entailed. Example: the predicate part of tp_2, `ex:hasSeniorManager`, matches the constant predicate map of triples map <#Departments>. There is nothing more we can deduce from this.
- If the triple pattern term is a variable, then the variable is bound to the constant value of the term map. This case is already taken care of in the projection part, that we illustrated above with triple pattern tp_3:

  ```
  genProjection(tp3, <#Staff>) = {ex:manages AS ?predicate}
  ```

  Thus again, no condition can be entailed.

(d) When a referencing object map is matched with a triple pattern term, a non-null condition must be added for each of the joined references to ensure that only valid values be joined. This is achieved by function *genCond* for the child triples map, and function *genCondParent* for the parent triples map. In the running example, the object of tp_1, variable `?dept`, is matched with the referencing object map of <#Staff>, whose join condition is:

```
rr:joinCondition [ rr:child "$.manages.*"; rr:parent "$.dept"]
```

This entails a non-null condition on the child reference in the first atomic query: `isNotNull($.manages.*), and` a non-null condition on the parent reference in the second atomic query: `isNotNull($.dept)`. Finally, we get the following conditions for tp_1:

```
genCond(tp1, <#Staff>, true) = {
               equals("Dunbar", $['lastname','familyname']),      // constant term
               isNotNull($.manages.*) }                           // join condition
genCondParent(tp1, <#Staff>, true) = {
               isNotNull($.dept),                                 // join condition
               isNotNull($.code) }                                // variable ?dept
```

(e) SPARQL filter. Cases (a) to (d) simply entail two types of condition: non-null or equality. SPARQL filters, on the other hand, have a much richer variety of functions and operators, including a subset of XQuery 1.0 and XPath 2.0. By construction, the SPARQL filter *f* passed as argument of $transTP_m$ mentions only variable of the triple pattern (ensured by function *sparqlCond*). The atomic abstract query keeps track of the filter using notation `sparqlFilter(f)`. If variables mentioned in the filter are matched with an xR2RML reference (a reference-valued term map or a template-valued term map), the corresponding xR2RML reference is provided in the *Project* part of the atomic query. Let us consider the short example below:

```
{ From    ← { … }
  Project ← { $.arrayField.* AS ?x }
  Where   ← { sparqlFilter(?x >= 5 && ?x < 10) }
}
```

The SPARQL filter "`?x >= 5 && ?x < 10`", and the *Project* part states that the values of ?x are generated by the xR2RML reference "$.arrayField.*". At the level of the abstract query language, the SPARQL filter and the projection are kept as is. The filter shall be translated into a target query in the subsequent translation step, *i.e.* when translating from the abstract query language to the target database query language.

Limit. The Limit part is a positive integer value representing the maximum number of results that the atomic query should return. It is provided in the atomic query with the incentive of limiting the size of intermediate results from inner queries.

> ***Running Example.*** We now summarize the way function *transTPm* computes abstract queries.
>
> The SPARQL basic graph pattern *bgp* consists of two triple patterns:
>
> tp_1 = `<http://example.org/staff/Dunbar> ex:manages ?dept`
> tp_2 = `?dept ex:hasSeniorMember ?senior.`
>
> Given the bindings for each triple pattern:
>
> $bind_m(bgp)$ = { (tp_1, {<#Staff>}), (tp_2, {<#Departments>}) },
>
> we can rewrite each triple pattern into an abstract query:
>
> $transTP_m(tp_2, true, \infty)$ =
>
> ```
> { From    ← { [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"] }
>   Project ← { $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior }
>   Where   ← { isNotNull($.code), isNotNull($.members[?(@.age >= 40)].name) }
> }
> ```
>
> and

```
transTP_m(tp_1, true, ∞) =
    { From    ← { [xrr:query "db.staff.find({})"] }
      Project ← { $.manages.* }
      Where   ← { equals("Dunbar", $['lastname', 'familyname']),
                  isNotNull($.manages.*) }
    } AS child
    INNER JOIN
    { PFrom    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      PProject ← { $.dept, $.code AS ?dept }
      PWhere   ← { isNotNull($.dept), isNotNull($.code) }
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

6.7 Abstract Query Optimization

At this point, the method we have exposed translates a SPARQL graph pattern into an effective abstract query, *i.e.* that preserve the semantics of the SPARQL query. Yet, shortcomings such as unnecessary complexity or redundancy may lead to the generation of inefficient queries, and in turn entail poor performances. Although we may postpone the query optimization to the translation into a concrete query language, it is interesting to figure out what optimizations can be done on the abstract representation first, and leave only database-specific optimizations to the subsequent stage.

SPARQL-to-SQL methods proposed various query optimizations [Unbehauen et al., 2013b; Rodríguez-Muro & Rezk, 2015; Elliott et al., 2009; Sequeda & Miranker, 2013]. In this section, we review some of these techniques, referring to the terminology defined in [Unbehauen et al., 2013b]. We show how these optimizations can be relevantly adapted to fit in the context of our abstract query language. In particular, we show that our translation method implements some of these optimizations by construction. In addition, we propose a new optimization, the *Filter Propagation*, that, to our knowledge, was not proposed in any SPARQL-to-SQL rewriting methods.

Examples of this section are provided based on the MongoDB example. Yet again, recall that these optimizations apply at the abstract query level and, consequently, they are generic and may apply with any other target database.

6.7.1 Filter Optimization

In a naive approach, the management of template strings can lead to inefficient target queries. Typically, when the translation of an R2RML template relies on the SQL string concatenation, a SPARQL query can be rewritten into something like this:

```
SELECT … FROM … WHERE (‘http://domain/’ || TABLE.ID) = ‘http://domain/1’
```

Such a query returns the expected results, but it is likely to perform very poorly: due to the concatenation, the query evaluation engine cannot take advantage of existing database indexes. Conversely, a much more efficient query would be:

```
SELECT (‘http://domain/’ || TABLE.ID)… FROM … WHERE TABLE.ID = 1
```

In our approach, equality conditions generated by the *genCond* and *genCondParent* functions apply to xR2RML references rather than on template-generated values, hence the *Filter Optimization* is enforced by construction.

6.7.2 Filter pushing

As we have mentioned in section 6.4, the translation of a SPARQL filter into an encapsulating SELECT-WHERE clause makes inner queries selectivity low, and the query evaluation process may have to deal with unnecessarily large intermediate results. In our approach, *Filter pushing* is enforced by construction by the *sparqlCond* function: relevant SPARQL filters are pushed down, as much as possible, in the translation of individual triple patterns.

6.7.3 Self-Join Elimination

The self-join issue has been investigated for R2RML-based SPARQL-to-SQL translation [Elliott et al., 2009; Unbehauen et al., 2013b]: it occurs when a relational table is joined with itself. We generalize this in our xR2RML-based translation: a self-join may occur when two triples maps, bound to two joined triple patterns, share the same logical source (`xrr:query` and `rml:iterator`). The atomic abstract queries Q_1 and Q_2 representing the two triple patterns are in an eliminable self-join situation when the following conditions are met:

(a) Both queries have the same *From* part, *i.e.* they refer to the same logical source, or one logical source is a subset of the other.
(b) They have at least one shared variable on which the join is to be performed.
(c) Both queries project the same xR2RML data element reference(s) as the same shared variable(s), *e.g.* if the xR2RML reference `"$.x"` is projected as variable `?x` in the left query, then the same projection must exist in the right query for the join to be an eliminable self-join. On the contrary, if projections are different: `"$.x1 AS ?x"` in Q_1 and `"$.x2 AS ?x"` in Q_2, then this is a regular self-join.
(d) Each reference projected as a shared variable must uniquely identify a document within query results: if Q_1 and Q_2 both have projection `"$.x AS ?x"`, then the xR2RML reference "`$.x`" must be declared as unique in at least one of the logical sources:

```
xrr:logicalSource [ xrr:query "…"; xrr:uniqueRef "$.x" ]
```

The *Self-Join Elimination* consists in merging both atomic queries into a single one, wherein the *Project* part merges the *Project* parts of both queries, and the *From* part is the most specific of the two *From* parts.

Condition (d) is illustrated in Figure 11 : on both sides, Q1 and Q2 depict the result of the atomic queries. Since they have the same *From* part, Q1 and Q2 actually contain the same results. On the left, two documents have a field x with value 1. Yet, x is not unique: several documents may have field x=1. Therefore, the self-join must not be eliminated as this is a regular self-join: documents with id 4 and 5 can be joined together on variable ?x. Conversely, on the right the logical source of one of the atomic queries declares the xR2RML reference "`$.x`" as unique with property `xrr:uniqueRef`. It follows that two documents with x=1 are necessarily the exact same document, and the resulting self-join can be eliminated.

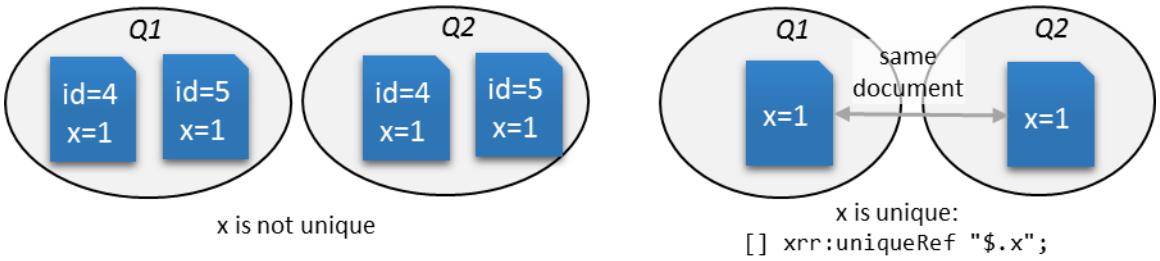


Figure 11: Self-join elimination on a unique xR2RML reference

Note: the `xrr:uniqueRef` property has been added to the xR2RML mapping language to enable the self-join elimination at the abstract query level. In conventional SPARQL-to-SQL approaches, it is generally not necessary as the schema does provide metadata about such unique field. Typically, the SPARQL-to-SQL query-rewriting engine inspects the database schema, looking for primary key or unicity constraints. On the contrary, with schema-less databases like MongoDB, unicity constraints are not made explicit, thus they must be stated declaratively within the mapping.

Running Example. The translation of tp_1 and tp_2 entails the following abstract query:

```
transₘ(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept}
      Where   ← {isNotNull($.code), isNotNull($.dept)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
    INNER JOIN
    { From    ← [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]
      Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← { isNotNull($.code), isNotNull($.members[?(@.age>=40)].name)}
    ON {?dept}
```

First, we change the natural left-to-right joins processing order: we embed the 2nd and 3rd atomic queries in curly brackets.

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    {
        { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
          Project ← {$.dept, $.code AS ?dept}
          Where   ← { isNotNull($.code), isNotNull($.dept)} }
        INNER JOIN
        { From    ← [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]
          Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
          Where   ← { isNotNull($.code), isNotNull($.members[?(@.age>=40)].name) }
        ON {?dept}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

The 2nd and 3rd atomic queries have the same *From* part, they are joined on variable `?dept`, variable `?dept` has the same projection in both queries: "`$.code AS ?dept`", and finally, that xR2RML reference "`$.code`" is declared as unique in both logical sources. Hence, this self-join can be eliminated.

We perform the self-join elimination by merging the two queries together: we merge the *Project* parts on the one hand, and the *Where* parts on the other hand. We obtain the following optimized abstract query:

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']), isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept),
                 isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

6.7.4 Self-Union Elimination

The UNION operator of the abstract query language can either be created when translating the SPARQL UNION operator, or during the translation of a triple pattern to which several triples maps are bound (in function $transTP_m$). Similar to the *Self-Join Elimination*, a union of several atomic abstract queries can be merged into a single one when they have the same *From* part, i.e. they share the same logical source.

The resulting atomic abstract query Q merges atomic abstract queries Q_1 and Q_2 as follows:

- The *Project* part of Q is the union of the *Project* parts of Q_1 and Q_2.
- The *Where* part of Q must be a condition that allows either the conditions of Q_1 or the conditions of Q_2, or both. Toward that end, we introduce the new condition operator *OR*. The *Where* part of Q is defined as: $OR(Q_1.Where, Q_2.Where)$.

6.7.5 Constant Projection

The *Constant Projection* optimization detects cases where the only projected variables in the SPARQL query are matched with constant values in the bound triples maps. In the relational database context, it has been referred to as the *Projection Pushing* optimization [Unbehauen et al., 2013b]. Nevertheless, we find this term somehow unintuitive, and we prefer the term *Constant Projection*.

Let us consider the example query below:

```
SELECT DISTINCT ?p WHERE {?s ?p ?o}.
```

In a naive approach, all triples maps of the mapping graph are bound to the triple pattern "`?s ?p ?o`". Hence, the resulting abstract query is a union of the atomic queries derived from all the triples maps in the mapping graph. In other words, this query will materialize the whole database before it can provide an answer. Besides, this kind of query is critical since it is typical of schema exploration queries.

Very frequently, the xR2RML predicate maps are constant-valued: the predicate is not computed from a database value, on the contrary it is defined statically in the mapping. This is typically the case in our running example that has only constant predicate maps defined by: "`rr:predicate ex:hasSeniorMember`" and "`rr:predicate ex:manages`". In such cases, given that the SPARQL query retrieves only DISTINCT values of the predicate variable ?p, no query needs to be run against the database at all: it is sufficient to collect the distinct constant values that variable ?p can be matched with.

More generally, this optimization checks if the variables projected in the SPARQL query are matched with constant term maps. If this is verified, the SPARQL query is rewritten such that the values of the projected variables be provided as an inline solution sequence using the SPARQL 1.1 VALUES clause. Following up on the example above, we would rewrite the query in this way:

```
SELECT DISTINCT ?p WHERE { VALUES ?p ( _:prop1 _:propr2 … )}.
```

The latter query can be evaluated without requiring any query to the target database.

6.7.6 Filter Propagation

We identified another type of optimization that was not implemented in the SPARQL-to-SQL context. This optimization applies in the inner join or left outer join of two atomic queries, and seeks to narrow down one of the joined queries by propagating filter conditions from the other query.

In an inner join, if the two queries have shared variables, then *equals* and *isNotNull* conditions of one query on those shared variables can be propagated to the other query. In a left join, propagation can happen only from right to left query since null values must still be allowed in the right query.

Example. Assume that abstract query Q is defined as the inner join of atomic queries Q_1 and Q_2:

```
{ From    ← { … }                        # Q1
  Project ← {$.field1 AS ?x}
  Where   ← {equals("value", $.field1)}
}
INNER JOIN
{ From    ← { … }                        # Q2
  Project ← {$.field2 AS ?x}
  Where   ← {}
} ON { ?x }
```

The equals condition in Q_1's *Where* part applies to an xR2RML reference that happens to be the reference projected as ?x. In other words, the join will only select documents from Q_1 and Q_2 where variable ?x equals "value".

In the right query, Q_2, another xR2RML reference is projected as ?x: "`$.field2`". Thus, the join will only match documents of Q_2 where the reference "`$.field2`" also returns "value". Consequently, we can add (propagate) a new condition to the *Where* conditions of Q_2: `equals($.field2, "value")`. We end up with the optimized query:

```
{ From    ← { … }
  Project ← {$.field1 AS ?x}
  Where   ← {equals("value", $.field1)}
}
INNER JOIN
{ From    ← { … }
  Project ← {$.field2 AS ?x}
  Where   ← { equals("value", $.field2)}
} ON { ?x }
```

Consequently, query Q_2 is more selective and the join can be computed faster.

6.8 Consolidated Running Example

To wrap this chapter up, in this section we put together all the steps of our running example.

Running Example. The SPARQL basic graph pattern *bgp* consists of two triple patterns:

```
tp1 = <http://example.org/staff/Dunbar> ex:manages ?dept
tp2 = ?dept ex:hasSeniorMember ?senior.
```

1. The $trans_m$ function rewrites *bgp* (section 6.4):

```
trans_m(bgp) =
     transTPm(tp1, true, ∞)
     INNER JOIN
     transTPm(tp2, true, ∞)
     ON {?dept}
```

2. We compute the bindings for each triple pattern (section 6.4.2):

```
bind_m(bgp) = { (tp1, {<#Staff>}), (tp2, {<#Departments>}) }
```

3. Based on the triple pattern bindings, we can rewrite each triple pattern into an abstract query (section 6.6):

```
transTPm(tp1, true, ∞) =
    { From    ← { [xrr:query "db.staff.find({})"] }
      Project ← { $.manages.* }
      Where   ← { equals("Dunbar", $['lastname','familyname']),
                  isNotNull($.manages.*) }
    } AS child
    INNER JOIN
    { PFrom    ← {[xrr:query "db.departments.find({})"]}
      PProject ← { $.dept, $.code AS ?dept }
      PWhere   ← { isNotNull($.dept), isNotNull($.code) }
    } AS parent
    ON child/$.manages.* = parent/$.dept

transTPm(tp2, true, ∞) =
    {  From    ← { [xrr:query "db.departments.find({})"] }
      Project ← { $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior }
      Where   ← { isNotNull($.code), isNotNull($.members[?(@.age >= 40)].name) }
    }
```

Thus:

```
transm(bgp, true) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']),
                 isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept}
      Where   ← {isNotNull($.code), isNotNull($.dept)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
    INNER JOIN
    { From    ← [xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]
      Project ← {$.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.members[?(@.age>=40)].name)}
    ON {?dept}
```

4. Lastly, we optimize the query by eliminating the self-join between the second and third atomic queries (section 6.7):

```
transm(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']),
                 isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept),
                 isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

6.9 Conclusion and Perspectives

The method presented in this chapter aims at fostering the development of SPARQL interfaces to heterogeneous databases, as we believe this is a key to the advent of the Web of Data. Leveraging R2RML-based SPARQL-to-SQL works, our method translates a SPARQL query into a pivot abstract query, utilizing xR2RML to describe the mapping of a target database to RDF. The method determines a set of relevant mappings for each SPARQL triple pattern, this set is reduced with respect to the join constraints and SPARQL filters, and *Atomic* A*bstract Queries* are determined by matching each triple pattern with the relevant xR2RML mappings. Lastly, query optimization techniques are enforced in order to produce an efficient abstract query and facilitate the subsequent translation into the target query language.

In the next chapter, we demonstrate the effectiveness of the method exposed here, taking the MongoDB document store as a target database. Before that, below we highlight several limitations of our method and discuss possible future works.

SPARQL support. At this point, SPARQL named graphs are not considered in the translation. However, it would be relatively easy to extend the method that computes triple pattern bindings in order to match named graphs (FROM, FROM NAMED) with xR2RML graph maps.

Abstract Query Optimization. The management of SPARQL filters is delegated to the translation into the target query language, using the *sparqlFilter* condition. Yet, some types of filter may be dealt with at the abstract query level, in order to alleviate the work required in the translation towards the target query language. For instance, operator BIND may be turned into equivalent *equals* conditions, and *bound* into *isNotNull* conditions.

Beyond the optimizations we have implemented, further query optimization challenges shall arise in order to develop an efficient query-processing engine. For instance, what is the most efficient order to compute INNER JOINs? In this regard, the query processing engine may need to embark query plan optimization logics such as the bind join [Haas et al., 1997] to inject intermediate results into a subsequent query, and the join re-ordering based on the number of results that queries shall retrieve, very similarly to the methods applied in distributed SPARQL query engines [Schwarte et al., 2011; Görlitz & Staab, 2011; Macina et al., 2016].

Support of xR2RML mixed-syntax paths. Although mixed-syntax paths are useful to materialize RDF terms from database values with embedded formats, they can lead to undecidable situations in the SPARQL rewriting context. For instance, in section 8.4.1 we translate a MongoDB database wherein JSON documents have a field `nomVernaculaire` whose value is a comma-separated string. The predicate-object map below builds object terms by selecting the first value (at index 0: `CSV(0)`) of the coma-separated string:

```
[] rr:predicateObjectMap [
     rr:predicate txrp:vernacularName;
     rr:objectMap [ xrr:reference "JSONPath($.nomVernaculaire)/CSV(0)" ] ]
```

Let us consider a SPARQL query containing the following triple pattern:

```
?vern txrp:vernacularName "Delphinus delphis".
```

The rewriting process will create an atomic abstract query wherein a condition matches value "`Delphinus delphis`" with the mixed-syntax path expression:

```
equals(JSONPath($.nomVernaculaire)/CSV(0), "Delphinus delphis")
```

Unfortunately, there is an infinite number of CSV strings where the first value is "Delphinus delphis". Hence, in this specific context, assuming that a `startsWith` function exists, we could rewrite the condition into something like:

```
startsWith($.nomVernaculaire, "Delphinus delphis")
```

Now, let us examine what happens if the field content is not a CSV value like in this example, but an XML snippet. We may end up with very complex expressions, for instance:

```
equals(JSONPath($.field)/XPath(//root/element[@type="some type"]), "Value")
```

Strikingly, we can imagine that translating such a condition into a target language may be difficult. More generally, it occurs that this problem amounts to deal with an arbitrary combination of JSONPath, XPath and CSV expressions. Although we may find solutions to this issue in specific situations, it seems illusory to seek a generic solution. Consequently, our SPARQL rewriting method does not deal with mixed-syntax paths. Nevertheless, in the context of custom functions written using CSVW and R2RML-F (as we suggested in section 5.8), it would be possible to define a transformation function along with an inverse transformation function, similar to the R2RML `rr:inverseExpression` property. Thus, the inverse transformation would be delegated to a function that embeds domain knowledge.

Chapter 7. SPARQL Access to MongoDB Documents

7.1 Introduction

In the previous chapter, we have exhibited an abstract query model and a method to translate a SPARQL query into an optimized abstract query, relying on xR2RML to describe the mapping of a target database to RDF. In this chapter, we now want to demonstrate that it is possible to translate an abstract query into an example target query language. More specifically, our goal is to bring NoSQL databases to the Web of Data. Thereby, we wish to illustrate the effort it takes to translate from SPARQL towards a query language that is far less expressive than SPARQL. Among the manifold existing NoSQL systems today, we consider the usual coarse categories of key-value stores, extensible column stores and documents stores[78]. Key-value stores contain data pairs accessed with very limited query languages, consisting essentially of two methods: `get(key)` and `put(key, value)`. They are designed for fast retrieval of indexed data, for instance as caching systems, but they are not meant for rich querying. Thus, in the absence of specific use case, the interest of translating such data into RDF is unclear. Extensible column stores provide a much richer data model and query language. Yet, their columnar data model makes them easily comparable with relational systems, and the added value of xR2RML in that case may not be obvious enough. Therefore, we feel like document stores are more suited to illustrate our method. Below, we describe the reasons for our choice in that matter.

Choice of the MongoDB Database. In recent years, the MongoDB[79] NoSQL document store has become a very popular actor in the NoSQL market. Some indicators confirm this popularity: at the time of writing, it is ranked in DB-Engines[80] as the first most popular NoSQL database, and the fifth if we consider relational databases. This ranking considers multiple inputs: number of results on major search engines, frequency of discussions in technical Web sites, job offerings (*e.g.* Indeed[81]), professional network profiles (*e.g.* LinkedIn), etc.

Although MongoDB was initially designed to deal with large distributed data sets, its popularity suggests that it is increasingly adopted as a general-purpose database. For instance, the Phenome project[82] stores phenotyping data in various MongoDB instances [Le Ngoc et al., 2016], small or big. Huma-Num[83], the French node of the DARIAH-EU[84] project, is a large infrastructure providing researchers in Digital Humanities with a portfolio of services including, among others, the hosting and

[78] We leave graph stores out of the discussion. Indeed, these are meant to store interconnected graphs and often provide a SPARQL or SPARQL-like interface. Considering one of them would probably not be a “striking” illustration.
[79] https://www.mongodb.org/
[80] http://db-engines.com/en/system/MongoDB
[81] http://www.indeed.com/jobtrends/mongodb,mongo,cassandra,hbase,couchdb,couchbase,membase,redis.html
[82] French Plant Phenomic Network (FPPN): https://www.phenome-fppn.fr/phenome_eng/
[83] Huma-Num: http://www.huma-num.fr/
[84] DARIAH-EU: http://www.dariah.eu/

maintenance of MongoDB databases. It is also noteworthy that MongoDB is provided as a service by several cloud service providers including Amazon EC2[85] and Microsoft Azure[86].

Therefore, beyond the hype and the controversy arguments about possible architecture flaws, MongoDB's success must be acknowledged. It is likely that many MongoDB instances host valuable data about all sorts of topics, that could benefit a large community at the condition of being made accessible as Linked Open Data. These elements motivated us to regard MongoDB as a good candidate to demonstrate the effectiveness of our method.

Related Works. So far, although the first version of MongoDB was developed in 2007, very little works have considered the idea of accessing MongoDB documents with SPARQL. Tomaszuk proposed a solution to use MongoDB as an RDF triple store [Tomaszuk, 2010]. The translation of SPARQL queries that he proposed is closely tied to the data schema and does not fit with arbitrary documents. MongoGraph[87] is an extension of the AllegroGraph triple store to query arbitrary MongoDB documents with SPARQL. But similarly to the Direct Mapping [Arenas et al., 2012], the approach comes up with an ad-hoc ontology (e.g. each JSON field name is turned into a predicate) and hardly supports the reuse of existing ontologies. More in line with our work, Botoeva et al. recently proposed a generalization of the OBDA principles to MongoDB [Botoeva et al., 2016a]. They describe a two-step rewriting process of SPARQL queries into the MongoDB *aggregate* query language. In section 7.6, we analyze in further details the relationship between their approach and ours.

In this chapter, we first describe the MongoDB query language along with an abstract representation of MongoDB queries (section 7.2), and the JSONPath language (section 7.3) that we use to express xR2RML data element references in the context of MongoDB. Then, we show that, far from being just a formality, the translation from the Abstract Query Language towards MongoDB is a challenging step, notably because of the discrepancy between the expressiveness of SPARQL and that of the MongoDB query language (section 7.4). This shall consist of three steps: (i) the translation of a projection from an Atomic Abstract Query into an abstract MongoDB projection argument (section 7.4.1), (ii) the translation of a condition from an Atomic Abstract Query into an abstract MongoDB query argument (section 7.4.2), and (iii) the rewriting and optimization of the abstract representation of a MongoDB query into a union of executable MongoDB queries (section 7.4.3). Finally, we propose an algorithm that orchestrates all the steps, from the SPARQL query until the generation of the RDF triples that match the SPARQL graph pattern (section 7.5). The last two sections discuss the overall approach and provide leads for future works.

7.2 The MongoDB Query Language

The MongoDB database comes with a rich set of APIs to allow applications to query a database in an imperative way. In addition, the MongoDB interactive interface defines a JSON-based declarative query language consisting of two query methods. The *find* method retrieves documents matching a set of conditions and returns a cursor to the matching documents. Optional modifiers amend the query

[85] https://www.mongodb.com/partners/amazon

[86] https://docs.mongodb.com/ecosystem/platforms/windows-azure/

[87] MongoGraph : http://franz.com/agraph/support/documentation/4.7/mongo-interface.html

to impose limits and sort orders. Alternatively, the *aggregate* method allows for the definition of processing pipelines: each document of a collection passes through the stages of a pipeline that creates a new collection, and so on. This allows for richer aggregate computations but comes with a much higher resource consumption that entails more unpredictable performance issues. Thus, as a first approach, this work considers the *find* query method, hereafter called the *MongoDB query language*. We describe below the language as defined in the MongoDB Manual v3.2[88]. In the last section of this chapter, we discuss a possible mix of the *find* and *aggregate* methods.

7.2.1 MongoDB *Find* Query Method

The MongoDB *find* query method takes two arguments:

(1) The query parameter is a JSON document that describes conditions about the documents to search for in the database. Specific query operators are denoted by a heading '$' character. Here are a few examples:
   - `{"decade":{$exists:true}}`: matches all documents with a field "decade".
   - `{"person.age":{$gte:18}}`: matches all documents with a field "person" whose value is a document having a field "age" whose value is 18 or more.
   - `{"staff.0.role":{$eq:"manager"}}`: matches all documents with an array "staff" whose first element (at index 0) has a field "role" with value "manager".
   - `{"staff":{$elemMatch:{"role":"developer"}}}`: matches all documents with an array "staff" in which at least one element is a document having a field "role" with value "developer".

(2) The optional projection parameter specifies the fields from the matching documents to return. In the example below, collection "people" is searched for documents with a field "age" whose value is 18 or more, but only the "name" field of each matching document is returned.

```
db.people.find({"age":{$gte:18}}, {"name": true})
```

The MongoDB documentation provides a rich description of the *find* query that however lacks precision as to the formal semantics of some operators. For instance, the query `{$or:[{"p.q":10},{"p.q":11}]}` retrieves documents where field "p" is a document having a field "q" whose value is either 10 or 11. We may be tempted to write the same query in another way: `{"p": {$or: [{"q":10},{"q":11}]}}`. This query is invalid though. It is unclear in the documentation why the `$or` and `$and` operators cannot be used as a condition on a field, but have to be at the top-level of the query document, or nested in an `$elemMatch`, `$and` or `$or` operator. Recently, attempts were made to clarify this semantics while underlining some limitations and ambiguities: Botoeva et al. [Botoeva et al., 2016b] mainly focus on the *aggregate* query and ignore some of the operators we use in our translation, such as `$where`, `$elemMatch`, `$regex` and `$size`. On the other hand, Husson [Husson, 2014] describes the *find* query, yet some restrictions on the operator `$where` are not formalized.

Therefore, in Definition 15 we specify the subset of the query language that we consider in our approach, and we underline some limitations and ambiguities. Operator keywords are bold, square brackets ('[', ']'), curly brackets ('{', '}') and characters ":", ",", "/" and "." are part of the language.

[88] https://docs.mongodb.com/v3.2/tutorial/query-documents/

Parenthesis groups "(...)", characters “*”, “+” and “|” are the conventional syntactic notation for occurrences and alternatives.

Definition 15. Grammar of a subset of the MongoDB query language

```
TOP_LEVEL_QUERY ::= {} |
                    { QUERY (, QUERY)* (, WHERE_QUERY)* } |
                    { WHERE_QUERY (, WHERE_QUERY)* }
QUERY           ::= FIELD_QUERY | OR_QUERY | AND_QUERY
FIELD_QUERY     ::= PATH: {OP: LITERAL} |
                    PATH: {$elemMatch: {QUERY(, QUERY)*}} |
                    PATH: {$regex: /REGEX/}
OP              ::= $eq | $ne | $lt | $lte | $gt | $gte | $size
OR_QUERY        ::= $or: [{QUERY} (, {QUERY})+]
AND_QUERY       ::= $and: [{QUERY} (, {QUERY})+]
PATH            ::= "(FIELD_NAME|ARRAY_INDEX)(.(FIELD_NAME|ARRAY_INDEX))*"
WHERE_QUERY     ::= $where: JS_BOOL_EXP
LITERAL         ::= literal value possibly in double quotes,
                    including specific values null, true, false
FIELD_NAME      ::= valid JSON field name
ARRAY_INDEX     ::= positive integer value
JS_BOOL_EXP     ::= valid JavaScript boolean expression
REGEX           ::= Perl-compatible regular expression

ARRAY_SLICE     ::= {PATH: {$slice: <nb_of_elts>}} |
                    {PATH: {$slice: [<skip>,<limit>]}}
```

A comma-separated sequence of QUERY elements (in the top-level query and in the `$elemMatch` operator) implicitly denotes a logical AND on the QUERY elements. Similarly, the `$and` operator denotes a logical AND on an array of QUERY elements. The `$and` operator is necessary when the same field name or operator has to be specified more than once. For instance, to select documents with a field “age” between 18 and 30, the query “`{"age":{$gte:18}, "age":{$lt:30}}`” is invalid since a JSON document cannot have two fields with the same name. This case requires using the $and operator: “`$and: [{"age":{$gte:18}}, {"age":{$lt:30}}]`”.

The `$elemMatch` operator (in FIELD_QUERY) matches documents with an array field in which at least one element matches all the specified QUERY criteria.

The `$where` operator (WHERE_QUERY) passes a JavaScript expression or function to the query system. It provides greater flexibility than other operators do. However, the JavaScript evaluation cannot take advantage of existing indexes and requires the database to process the JavaScript expression for each document. This issue can seriously hinder performances, and MongoDB strongly recommends to use `$where` only when the query cannot be expressed using another operator. The `$where` operator is valid only in the top-level query document: it cannot be used inside a nested query such as the `$elemMatch` operator. This restriction makes a strong difference with SQL, and has a major impact on the rewriting process.

The ARRAY_SLICE definition is separated from the above ones, as an array slice does not apply in the query part but in the projection part of a MongoDB *find* query. For instance, query

```
db.collection.find({comments:{$size: 100}}, {comments:{$slice: 5}})
```

selects documents that have an array “comments” with 100 elements, and projects only the first five elements.

7.2.2 Semantics Ambiguities

The MongoDB query language allows ambiguous short-cut expressions to name paths in the JSON documents. For instance, query `{"p":{$eq:3}}` matches documents where p is a field with value 3, such as `{"p":3}`. Surprisingly, it also matches documents where p is an array wherein at least one element has value 3, *e.g.* `{"p":[3,4]}`. But the latter document would equally be matched by query `{"p":{$elemMatch:{$eq:3}}}`. This gets even worse with a sequence of field names, as each field name may be considered for what it is, *i.e.* exactly one field, or as a short-cut for the elements of an array field. With this logic, query `{"p.q":{$eq:3}}` matches several types of documents depending on how we interpret p and q, such as `{"p":{"q":3}}`, `{"p":[{"q":3}]}` and `{"p":[{"q":[3]}]}`.

Consequently, given the ambiguous notation of the MongoDB query language, it is hardly possible to write a MongoDB query whose semantics would be provably equivalent to a SPARQL query. Let us further elaborate on the possible impact of such ambiguities on the query translation process. The xR2RML mapping of a MongoDB database is written with a given schema in mind: the mapping designer expects documents to follow it, although the database is schemaless; this schema is implied by the JSONPath expressions that he/she embeds in the mapping. In our translation method, we do not use shortcut notations, instead we only use the most specific notation, *e.g.* an array element is always queried using the `$elemMatch` operator. Thus, it is likely that the rewriting we come up with will indeed retrieve only the expected documents. Yet, we cannot ignore the possibility that the database contains other documents, with a somehow different schema, that shall also be retrieved by the query due to the ambiguous notation even though this was not in the mapping designer’s intention.

> ***Example.*** We consider a MongoDB collection where documents are shaped like this one:
>
> ```
> {"p": { "q":3, "r":4 } }
> ```
>
> An xR2RML mapping designer may use JSONPath expression `$.p.q` to get the value of q. When matching the xR2RML mapping with a SPARQL triple pattern, we may come up with an atomic abstract query where a condition is:
>
> ```
> equals($.p.q, 3)
> ```
>
> This condition shall be translated into the MongoDB query below:
>
> ```
> db.collection.find({"p.q":{$eq: 3}})
> ```
>
> This query indeed matches the document shown above, but it may also match somewhat different documents, although this was not the intention of the mapping designer. For instance, in the document below, the value of “p” is no longer a document but an array field, whereof elements are documents with a field “q”:
>
> ```
> { "p": [ {"q":3}, … ] }
> ```
>
> Such additional documents, not considered during the mapping design, may lead to the generation of unpredictable and possibly erroneous RDF triples.

Unfortunately, there does not seem to be a method ensuring that such flaw cannot occur. Yet, a good practice to limit the likelihood thereof is to filter irrelevant documents at the earliest stage. In the example above, we know that field “p” may be used as both a document field and an array field. Therefore, the query in the logical source should try to rule out documents where “p” is an array:

```
[] xrr:logicalSource [
    xrr:query "db.collection.find({'p': {$not:  {$type: 'array'}}})"
];
```

7.2.3 Abstract Representation of a MongoDB Query

As we shall show in the next sections, the production of a target MongoDB query is not a straightforward process: it involves the iterative construction and optimization of separate snippets of query. To facilitate the process, we define an *abstract representation of a MongoDB query*. It is a technical artefact allowing for handy query manipulation and transformation. We sometimes refer to this abstract representation as an *abstract MongoDB query*, however it must not be confounded with the *Abstract Query Language* that we defined in Chapter 6.

Definition 16 lists the clauses of this representation as well as their translation into a concrete query string, when relevant.

In the COMPARE clause definition, `<op>` stands for one of the MongoDB comparison operators: `$eq`, `$ne`, `$lte`, `$lt`, `$gte`, `$gt`, `$size` and `$regex`. Let us consider the following example abstract query:

```
AND( COMPARE(FIELD(p) FIELD(0), $eq, 10), FIELD(q) ELEMMATCH(COND(equals("val")) )
```

It matches all documents where “p” is an array field whose first element is 10, and “q” is an array field in which at least one element has value “val”. Its concrete representation is:

```
$and: [ {"p.0": {$eq:10}}, {"q": {$elemMatch: {$eq:"val"}}} ]
```

The NOT_SUPPORTED clause helps keep track of any location, within the abstract query, where the condition cannot be translated into an equivalent MongoDB query element. It shall be used in the last rewriting and optimization phase.

The UNION clause is semantically equivalent to a logical OR. However, while the OR can be computed by the MongoDB database, the UNION shall be computed by the query-processing engine based on the result of queries `<query1>`, `<query2>`, etc. It may be produced during the last rewriting and optimization phase (see section 7.4.3).

Definition 16. Abstract Representation of a MongoDB query

AND(<expr$_1$>, <expr$_2$>, …)	→ **$and**:[<expr$_1$>,<expr$_2$>, …]
OR(<expr$_1$>, <expr$_2$>, …)	→ **$or**:[<expr$_1$>,<expr$_2$>, …]
WHERE(<JavaScript expr>)	→ **$where**:'<JavaScript expr>'
ELEMMATCH(<exp$_1$>,<exp$_2$>, …)	→ **$elemMatch**:{<exp$_1$>,<exp$_2$>, …}
FIELD(p_1) FIELD(p_2)... FIELD(p_n)	→ "$p_1.p_2….p_n$":
SLICE(<expr>, <number>)	→ <expr>:{**$slice**: <number>}
COND(equals(v))	→ **$eq**:v
COND(isNotNull)	→ **$exists**:**true, $ne**:**null**
EXISTS(<expr>)	→ <expr>:{**$exists**: **true**}
NOT_EXISTS(<expr>)	→ <expr>:{**$exists**: **false**}
COMPARE(<expr>, <op>, <v>)	→ <expr>:{<op>: <v>}
NOT_SUPPORTED	→ ∅
CONDJS(equals(v))	→ == v
CONDJS(equals("v"))	→ == "v"
CONDJS(isNotNull)	→ != null
UNION(<query$_1$>, <query$_2$>, …)	*Same semantics as OR, but processed by the query processing engine*

Note. This UNION clause must not be confused with the UNION operator of the abstract query language. The MongoDB UNION clause applies to a set of JSON documents retrieved from the database, whereas the abstract UNION operator applies to RDF triples.

7.3 The JSONPath Language

JSONPath[89] is a domain specific language designed to read, parse and extract data from JSON documents. It was proposed in 2007 by Stefan Goessner, as an analogy to the XPath[90] W3C standard. As of today, JSONPath is not a standard, however its definition remains stable and a large community provides and maintains implementations for various programming languages. Definition 17 describes the grammar of JSONPath. Bold characters ('$', '*', '.', '[', ']') are part of the language. Notice that characters "(" and ")" are part of the language in the FILTER and CALC_INDEX expressions, whereas in FIELD_ALT and INDEX_ALT expressions they simply denote groups. Similarly, the "*" character is part of the language in expression WILDCARD, but denotes 0 to any occurrences in other expressions.

Let us give a few illustrating examples:

- `$.names.*`: selects all elements of array "names", like in: "`{"names":["mark","john"]}`", or all fields of sub-document "names", like in "`{"names":{"firstname":"mark","lastname":"john"}}`".
- `$.books[1,3]`: selects the second (index 1) and fourth (index 3) elements of array "books".
- `$.books[1:3]`: selects all books from index 1 (inclusive) until index 3 (exclusive), *i.e.* at indexes 1 and 2.
- `$.books[(@.length - 1)]` or `$.books[-1:]`: select the last element of array "books". In the "[()]" notation, "@" refers to the parent element "books".

[89] http://goessner.net/articles/JsonPath/

[90] http://www.w3.org/TR/1999/REC-xpath-19991116/

- `$.team[?(@.members <= 10)].name`: select the name of teams that have 10 members or less, *i.e.* "team" is an array, among its elements we select those that have a field "members" whose value is 10 or less, and finally we select the field "name" of those elements. Unlike the aforementioned example, in the "[?()]" notation "@" refers to elements of the array.
- `$..author`: selects all "author" fields anywhere in the document.

Definition 17. Grammar of the JSONPath language

```
JSONPATH        = $(WILDCARD | FIELD_NAME | ARRAY_INDEX | DESCENDANT | FIELD_ALT |
                  INDEX_ALT | ARRAY_SLICE | FILTER | CALC_INDEX)*
WILDCARD        = .*|[*]
FIELD_NAME      = FIELD_NAME_DOT | FIELD_NAME_BRKT
FIELD_NAME_DOT  = .<name>
FIELD_NAME_BRKT = ["<name>"]
ARRAY_INDEX     = [<int>]
DESCENDANT      = ..
FIELD_ALT       = ["<name>"(,"<name>")+]
INDEX_ALT       = [<int>(,<int>)+]
ARRAY_SLICE     = [<start>:<end>:<step>] | [<start>:<end>] | [<start>:]
FILTER          = [?(<script expression>)]
CALC_INDEX      = [(<script expression>)]
```

Note. JSON field names with special characters such as '#', '&' or '/' etc. are supported by MongoDB, but in JSONPath they require the bracket notation, e.g. ["field/#1"].

In an array slice, if the <start> is omitted it defaults to 0, *e.g.* `$.books[:2]` selects the first two books. If <end> is omitted, it defaults to the index of the last element of the array plus one (<end> index is exclusive). <start> and <end> can be positive (the index is counted from the start of the array), or negative (the index is counted from the end of the array), e.g. `$.books[-2:]` selects the last two books.

Restrictions on the usage of JSONPath expressions.

Script expressions. The FILTER expression filters elements of an array based on <script expression> that must evaluate to a Boolean. CALC_INDEX selects the element of an array at index <script expression> that evaluates to a positive integer. In both cases, the language definition says that <script expression> is written in "the syntax of the underlying script engine". This design choice has a strong shortcoming: it binds the language definition to its implementations, since the underlying script engine depends on the implementation, and in the worst case, there may even not be any underlying script engine at all. That made sense in the initial JavaScript implementation of Goessner, but this is subject to various interpretations in other implementations. For instance in the Java port[91] of Goessner's implementation, developers have chosen to implement a very limited subset of JavaScript.

In our rewriting approach, we stick to the idea that those expressions are JavaScript, keeping in mind that its support may vary depending on the JSONPath implementation that is being used.

[91] https://github.com/jayway/JsonPath

Wildcard semantics. In JSONPath, the wildcard '*' is equally applicable to arrays and documents. In an array, it stands for any element of the array, while in a document it stands for any field of the document. In MongoDB conversely, documents and arrays are not treated equally: the `$elemMatch` operator applies specifically to arrays, and it is not possible to match any field in a document (there is no equivalent of the '*' character for a document). Therefore, to be able to translate JSONPath expressions into MongoDB, we restrict the use of the wildcard to arrays, which is its most common usage anyway.

Filters. In the JSONPath reference, it is unclear whether the filter notation `[?(<script expression>)]` applies to arrays, or to arrays and documents. Some implementations allow both with somewhat confusing semantics, *e.g.* in the expression `$.p[?(@.q)]`:

- If “p” is an array field, then “@” refers to each of its elements, meaning that only elements where there exists a field “q” are matched. The drawback is that it is not possible to write a condition about an element given by its index. For instance, to match arrays in which the 11th element is 0, we would like to write `$.p[?(@[10] == 0)]`, which is invalid because, in that case, “@” should refer to the array p but not to its elements.
- Conversely, if “p” is a document, “@” refers to “p” itself, meaning that “p” matches only if it is a document with a field “q”.

Furthermore, some tests show that different implementations have made different interpretations in this matter. To get rid of any confusion, in this work we restrict the usage of filters “[?()]” to arrays. Therefore expressions like `$.p[?(…)]` shall be understood as this: “p” is an array field, the “@” character refers to its elements.

Root element of JSON documents. In MongoDB, the root element of a document cannot be an array, e.g. `["mark","john"]` is not a valid MongoDB document, but `{"people":["mark","john"]}` is valid. Consequently, the JSONPath expressions we consider must not start with elements denoting an array field. For instance, expressions "`$[0]`" and "`$[1,3,5]`" are invalid in our context. Additionally, given the aforesaid restriction on the wildcard, expressions starting with "`$.*`" or "`$[*]`" are not supported in our context either.

Descendent operator. Unlike JSONPath, MongoDB does not provide a descendent operator that would look for a pattern at any depth of the documents. Consequently, our rewriting method does no support JSONPath expressions using the “..” descendent operator.

7.4 Translation of an Abstract Query into MongoDB queries

Let us recall the components of the abstract query language. Operators INNER JOIN, LEFT OUTER JOIN and UNION are entailed by the dependencies between graph patterns of the SPARQL query. A UNION operator may also arise when a triple pattern is bound to more than one xR2RML mapping, and an INNER JOIN operator may be generated when a mapping denotes a cross-reference towards another

mapping. LIMIT and FILTER are solution modifiers, they bear the same semantics as in SPARQL, and we have seen that, by construction, they are pushed down into relevant atomic abstract queries as much as possible.

INNER JOIN, LEFT OUTER JOIN and UNION operators relate atomic abstract queries of the form *{From, Project, Where, Limit}*. The *From* part contains the mapping's logical source. The *Project* part lists the xR2RML references that should be projected, *i.e.* part of the result of the atomic abstract query. The *Where* part is calculated by matching triple pattern terms with relevant term maps; this shall generate either *isNotNull* conditions for SPARQL variables, *equals* conditions for triple pattern constant terms, or *sparqlFilter* conditions that encapsulate SPARQL filters. Finally, the *Limit* part denotes an optional maximum number of results.

To achieve a translation from the abstract query language towards the MongoDB query language considered in section 7.2, we must figure out which components of an abstract query have an equivalent MongoDB rewriting, and, conversely, which components shall be computed by the query-processing engine. Below, we analyze the possible situations.

Atomic Abstract Query. In the context of MongoDB, xR2RML data element references are JSONPath expressions. The translation of an atomic abstract query towards MongoDB amounts to translate (i) projections of JSONPath expressions into MongoDB projection arguments, and (ii) conditions on JSONPath expressions into equivalent MongoDB query operators. Besides, the *Limit* part of the atomic query shall limit the number of results returned by the MongoDB query.

INNER JOIN and LEFT OUTER JOIN. MongoDB does not support joins, as a consequence we cannot translate the INNER JOIN and LEFT OUTER JOIN operators into MongoDB equivalent queries. The query-processing engine will be in charge of joining the RDF triples generated by both joined queries.

UNION. The UNION case is trickier, and depends on the graph patterns to which it applies. Let us consider the following SPARQL graph pattern, where tp_n is any triple pattern:

```
{ tp1. tp2.  } UNION { tp3. tp4. }
```

Each member of the union is translated into an INNER JOIN computed by the query-processing engine. Consequently, since the members are not processed within MongoDB, the UNION operator cannot be processed within MongoDB either. The issue occurs likewise as soon as one of the members is either an INNER JOIN or LEFT OUTER JOIN.

Let us now consider the case where union members are atomic queries:

```
{ tp1. } UNION { tp2. }
```

is translated into the abstract query:

```
{ From1, Project1, Where1, Limit } UNION { From2, Project2, Where2, Limit }
```

Under some circumstances, we can translate the UNION into a MongoDB `$or` operator. For example:

```
{ db.collection.find({"q": $exists}), Project1, equals($.p1, value1), Limit }
UNION
{ db.collection.find(), Project2, equals($.p2, value2), Limit }
```

can indeed be translated into the MongoDB query:

```
db.collection.find({$or: [ { "p1":{$eq:value1}, "q":$exists },
                           { "p2":{$eq:value2} }
                         ]}, …)
```

This case works because both *From* parts (i) query the same collection, and (ii) can be nested beneath an `$or` operator. But if *e.g.* one of them contains a `$where` operator, then the resulting query is no longer valid.

Consequently, in a first approach, we always shift the processing of the UNION abstract operator to the query-processing engine. Further works could attempt to characterize more specifically the situations where a UNION can be processed within MongoDB.

FILTER and LIMIT. The FILTER and LIMIT SPARQL clauses are managed such that parts of them are pushed down into relevant atomic queries, in the form of a *sparqlFilter* condition in the *Where* part, or as the *Limit* part, respectively. Therefore, the FILTER and LIMIT abstract operators apply at the upper level, as modifiers of the result of UNION, INNER JOIN and/or LEFT OUTER JOIN operators. Hence, given that UNION and JOIN operators are not processed within MongoDB, the FILTER and LIMIT operators cannot be processed within MongoDB either.

Ultimately, it occurs that only the atomic abstract queries can be processed within MongoDB. Other abstract operators shall be computed by the query-processing engine. More generally, the translation consists of two steps depicted in Figure 12. First, we translate each projection of an atomic abstract query into a projection argument, and each condition into the abstract representation of a MongoDB query. Several shortcomings may appear at this stage, such as untranslatable JSONPath expressions or unnecessary complexity. Thus, in step 2 we rewrite and optimize this abstract MongoDB query into a union of valid, executable MongoDB queries.

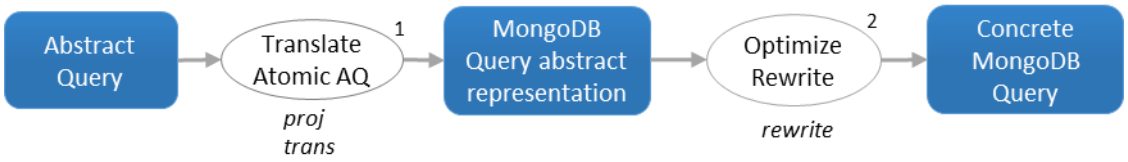


Figure 12: Translation from the Abstract Query Language to the MongoDB Query Language

The rest of this section describes these steps in further details. Below, we illustrate the expected result using the running example.

> ***Running Example.*** In section 6.8, we showed that the translation of tp_1 entails the atomic abstract query below:
>
> ```
> { From    ← { [xrr:query "db.staff.find({})"] }
>   Project ← { $.manages.* }
>   Where   ← { equals("Dunbar", $['lastname','familyname']),
>               isNotNull($.manages.*) }
> }
> ```
>
> The projection of the "`$.manages.*`" xR2RML reference is translated into the MongoDB projection argument: "`"manages": true`".

The condition `equals("Dunbar", $['lastname','familyname'])` is translated into a concrete MongoDB query as follows:

```
$or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}]
```

Similarly, the condition `isNotNull($.manages.*)` is translated into:

```
"manages": {$elemMatch: {$exists:true, $ne:null}}
```

Those conditions are used to augment the query of the *From* part, provided by the `xrr:query` and `rml:iterator` properties. In this regards, our example is trivial since the query in the logical source of triples map <#Staff> is empty ("{}") and there is no iterator.

Altogether, the atomic abstract query is translated in the MongoDB query:

```
db.staff.find(
       { $or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}],
         "manages": {$exists:true, $ne:null] },
       { "manages": true }
)
```

Limitations. In the current status of this work, we consider the translation of non-null and equality conditions into MongoDB. For the sake of simplicity, we do not consider the translation of SPARQL filters at this point. SPARQL 1.0 filters come with a broad set of conditional expressions including logical comparisons, literal manipulation expressions (string, numerical, boolean), XPath constructor functions, casting functions for additional datatypes of the RDF data model and SPARQL built-in functions (lang, langmatches, datatype, bound, sameTerm, isIRI, isURI, isBlank, isLiteral, regex), not to mention possible extension functions. Considering this broad scope of expressions within the translation towards MongoDB would yield a significant additional cumbersomeness without changing the translation principles. Yet, it remains clear that an implementation should consider filter expressions for the sake of completeness.

Conventions. The following notations are used in the subsequent sections:

- **<cond>**: is a condition on a JSONPath expression, that we want to translate into MongoDB query elements: it is written either *isNotNull* or *equals(<value>).*
- **<JP>**: denotes a (possibly empty) JSONPath expression.
- **<JP:F>**: denotes a non-empty JSONPath sequence of field names and array indexes, without the heading '$' character: *e.g.* ".p.q.r", ".p[10]["r"]".
- **<bool expr>**: denotes a JavaScript expression that evaluates to a boolean value.
- **<num expr>**: denotes a JavaScript expression that evaluates to a positive integer.

7.4.1 Translation of Projections

The *From* part of an atomic abstract query lists JSONPath expressions whose JSON fields should be part of the query results. Given the restrictions on JSONPath expressions defined in section 7.3, this section defines the recursive function *proj* that translates a JSONPath expression into the projection argument of a MongoDB *find* query. Function *proj* builds a list of paths that shall be projected. For instance, the

JSONPath expression "`$.p.q`" shall be translated into the abstract representation "`FIELD(p) FIELD(q)`" that, in turn, shall be translated into the concrete projection argument "`"p.q":true`".

Function *proj* implements a set of rules, depicted in Algorithm 2, that apply when the JSONPath expression matches a certain pattern. The JSONPath expression is checked against the patterns in the order of the rules. When a match is found, the rule is applied and the search for a match stops. Algorithm 2 uses function *projJS* defined later on in section 7.4.1.4

Algorithm 2: Translation of a JSONPath expression into a MongoDB projection argument (function proj(JSONPath expression))

P0 **proj**($<JP>) → **proj**(<JP>)

P1 **proj**(∅) → ∅

P2 *JavaScript filter and calculated array index*

(a) **proj**(<JP:F>[?(<bool_expr>)]<JP>) → **proj**(<JP:F>) × **projJS**(<bool_expr>, **proj**(<JP>))

(b) **proj**(<JP:F>[<num_expr>)]<JP>) → **proj**(<JP:F>) × **projJS**(<num_expr>, **proj**(<JP>))

P3 *Array expressions: wildcard (a, b), array index alternative (c), array slice (d)*

(a) **proj**(.*<JP>) → **proj**(<JP>)

(b) **proj**([*]<JP>,) → **proj**(<JP>)

(c) **proj**([i, j ,...]<JP>) → **proj**(<JP>)

(d) **proj**([<slice expression>]<JP>) → **proj**(<JP>)

P4 *Field name (a, b) or field alternative (c)*

(c) **proj**(.p<JP>) → **FIELD**(p) **proj**(<JP>)

(d) **proj**(["p"]<JP>) → **FIELD**(p) **proj**(<JP>)

(e) **proj**(["p","q",...]<JP>) → **FIELD**(p) **proj**(<JP>), **FIELD**(q) **proj**(<JP>), ...

The projection argument of a MongoDB *find* query is fairly simple: it consists of paths followed by "true" to project the path, or "false" to not project the path. Notations "true" and "false" cannot be mixed: either named paths are explicitly projected with "true" and unnamed paths are implicitly restricted, or paths are explicitly restricted with "false" and unnamed paths are implicitly projected. In our case, we use the "true" notation to explicitly list projected paths.

A path may be a single field name *e.g.* "p", or a sequence of field names. In the latter, two semantics can apply to the path "p.q":

- If the value of "p" is an embedded document, then "p" is projected along with only the field "q" of embedded documents;
- If the value of "p" is an array, then "p" is projected along with all its elements, but only the field "q" of its elements is projected.

We now describe the way a JSONPath expression is treated by each rule in Algorithm 2. Rule P2 is described last as it requires the understanding of other rules first.

7.4.1.1 Rules P0 and P1

Rule P0 is the entry point of the translation process. The heading "$" character is simply removed and the process goes on with the rest of the expression.

Conversely, rule P1 is the termination point, when the JSONPath expression has been fully parsed. The rule simply returns Ø to stop the process. Implicitly, all FIELD clauses generated beforehand are to be translated into a projection argument with the value "true".

7.4.1.2 Rule P3: Array Notations

Rule P3 deals with different types of array notations: wildcard, array index alternative and array slice. The MongoDB projection argument cannot select elements from an array: they are all projected, but we can specify which fields of the elements are projected. Thus, whether all elements are selected with the wildcard or only a subset with the alternative or slice, the translation simply goes on with parsing anything that comes after the array notation.

Examples: JSONPath expression "`$.p.*.q`" is translated into the argument "`"p.q":true`" that projects the field "q" of all elements of "p". Similarly, in JSONPath expressions "`$.p[1,3,5].q`" and "`$.p[2:4].q`", the alternative and slice notations are ignored. Both are translated into the same projection argument "`"p.q":true`".

7.4.1.3 Rule P4: Field Names

Rule P4 deals with regular field names, and simply adds a FIELD clause each time a new field name is encountered.

Example: JSONPath expression "`$.p.q.r`" shall be translated into the abstract representation "`FIELD(p) FIELD(q) FIELD(r)`", which shall in turn be rewritten into the projection argument "`"p.q.r": true`".

7.4.1.4 Rule P2: JavaScript Filter and JavaScript Calculated Array Index

JavaScript expressions, whether in a Boolean filter or a calculated index, are managed in a somewhat specific way. Let us take an example: JSONPath expression "`$.p[?(@.q && @.r)].s`" matches documents with an array field "p" whose elements have at least a field "q" and a field "r", and it returns the value of field "s", *e.g.* the document "`{"p": ["q":1, "r":2, "s":3]}`" matches the JSONPath expression and returns 3, *i.e.* the value of "s".

To project all appropriate fields, we must produce the projection argument:

```
"p.q":true, "p.r":true, "p.s":true
```

In other words, we have to make a product of the fields named before, inside and after the JavaScript filter. To that end, we define function *projJS*(<JS expr>, <FIELD clause>) that returns a list of FIELD clauses: one clause for each field named in the JavaScript (JS) expression, and the additional FIELD clauses provided as a second argument. Then, we denote by "×" the product between the FIELD clauses produced by either the *proj* or *projJS* functions.

For instance, rule P2(a) applies as follows:

proj(`$.p[?(@.q && @.r)].s`)

→ **proj**(.p) × **projJS**(`@.q && @.r`, **proj**(.s))

which is rewritten as follows:

→ FIELD(p) × (FIELD(q), FIELD(r), FIELD(s))

→ FIELD(p) FIELD(q), FIELD(p) FIELD(r), FIELD(p) FIELD(s)

This generates the three expected paths: “p.q”, “p.r” and “p.s”.

The management of calculated array indexes follows the same procedure.

7.4.1.5 Running Example

In section 6.8, we translated the SPARQL query of our running example in the optimized abstract query below:

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']),
                 isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept),
                 isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

In the first atomic query, projection “`$.manages.*`” is translated as follows (at each step, the rule that applies is named at the end of the line):

```
proj($.manages.*)
   -> proj(.manages.*)           (P0)
   -> FIELD(manages) proj(.*)    (P4)
   -> FIELD(manages) proj()      (P3)
   -> FIELD(manages)             (P1)
```

This abstract representation is finally translated into the simple projection argument:

```
"manages:true"
```

In the second atomic query, projections “`$.dept`” and “`$.code`” are trivial. Let us focus on the last one, “`$.members[?(@.age>=40)].name AS ?senior`”:

```
proj($.members[?(@.age>=40)].name)
   -> proj(.members[?(@.age>=40)].name)                    (P0)
   -> proj(.members) × projJS(@.age>=40, proj(.name))      (P2)
   -> FIELD(members) × projJS(@.age>=40, FIELD(name))      (P4)
   -> FIELD(members) × ( FIELD(age), FIELD(name) )         (def. of projJS)
   -> FIELD(members) FIELD(age), FIELD(members) FIELD(name)
```

This abstract representation gets translated into the projection argument:

```
"members.age":true, "members.name":true
```

7.4.2 Translation of Conditions

Given the subset of the MongoDB query language considered in section 7.2 and the restrictions on JSONPath expressions defined section 7.3, this section defines the recursive function *trans* that translates an abstract query condition on a JSONPath expression into the abstract representation of a MongoDB query. Function *trans* takes two arguments, one for the JSONPath expression and a second for the condition, such that:

- Condition *isNotNull(<JSP expr.>)* is translated into *trans(<JSP expr.>, isNotNull)*, and
- Condition *equals(<JSP expr.>, <value>)* is translated into *trans(<JSP expr.>, equals(<value>)).*

Function *trans* is detailed in Algorithm 3. It consists of a set of rewriting rules that apply when the JSONPath expression matches a certain pattern. The JSONPath expression is checked against the patterns in the order of the rules. When a match is found, the rule is applied and the search for a match stops. Before getting any further, we illustrate the approach using the running example.

Running Example. In section 6.8, we translated the SPARQL query of our running example in the optimized abstract query below:

```
trans_m(bgp, true, ∞) =
    { From    ← {[xrr:query "db.staff.find({})"]}
      Project ← {$.manages.*}
      Where   ← {equals("Dunbar", $['lastname', 'familyname']),
                 isNotNull($.manages.*)}
    } AS child
    INNER JOIN
    { From    ← {[xrr:query "db.departments.find({})"; xrr:uniqueRef "$.code"]}
      Project ← {$.dept, $.code AS ?dept, $.members[?(@.age>=40)].name AS ?senior}
      Where   ← {isNotNull($.code), isNotNull($.dept),
                 isNotNull($.members[?(@.age>=40)].name)}
    } AS parent
    ON child/$.manages.* = parent/$.dept
```

Let us focus on the conditions of the first atomic query.

Condition `isNotNull($.manages.*)` is translated using function *trans* as "`trans($.manages.*, isNotNull)`" that goes through the following steps:

- Rule R0 first matches an expression with a heading '$', it returns:

  ```
  trans(.manages.*, isNotNull)
  ```

- Rule R8 matches a simple field name and returns:

  ```
  FIELD(manages) trans(.*, isNotNull)
  ```

- Lastly, rules R7 and R1 translate `trans(.*, isNotNull)` into `ELEMMATCH(COND(isNotNull))`. This comes up with the abstract MongoDB query:

  ```
  FIELD(manages) ELEMMATCH(COND(isNotNull))
  ```

- Then, by applying Definition 16, we obtain the final concrete query:

```
"manages": {$elemMatch: {$exists:true, $ne:null}}
```

Following the same algorithm, condition "`equals("Dunbar", $['lastname','familyname'])`" is translated into the abstract MongoDB query:

```
OR( FIELD(lastname) COND(equals("Dunbar")),
    FIELD(familyname) COND(equals("Dunbar")))
```

In turn, it is translated into the concrete query:

```
$or: [{"lastname": {$eq: "Dunbar"}}, {"familyname": {$eq: "Dunbar"}}]
```

Algorithm 3: Translation of a condition on a JSONPath expression into an abstract MongoDB query (function trans(JSONPath expression, <cond>))

R0 **trans**($, <cond>) → **NOT_SUPPORTED**
trans($<JP>, <cond>) → **trans**(<JP>, <cond>)

R1 **trans**(∅, <cond>) → **COND**(<cond>)

R2 *Field alternative (a) or array index alternative (b)*
(a) **trans**(<JP:F>["p","q",...]<JP>, <cond>) →
OR(**trans**(<JP:F>.p<JP>, <cond>), **trans**(<JP:F>.q<JP>, <cond>), ...)
(b) **trans**(<JP:F>[i,j,...]<JP>, <cond>) →
OR(**trans**(<JP:F>.i<JP>, <cond>), **trans**(<JP:F>.j<JP>, <cond>), ...)

R3 *Heading field alternative (a) or heading array index alternative (b)*
(a) **trans**(["p","q",...]<JP>, <cond>) →
OR(**trans**(.p<JP>, <cond>), **trans**(.q<JP>, <cond>), ...)
(b) **trans**([i,j,...]<JP>, <cond>) →
OR(**trans**(.i<JP>, <cond>), **trans**(.j<JP>, <cond>), ...)

R4 *Heading JavaScript filter on array elements, e.g. $.p[?(@.q)].r*
trans([?(<bool_expr>)]<JP>, <cond>) →
ELEMMATCH(**trans**(<JP >, <cond>), **transJS**(<bool_expr>))
*(Function **transJS** is defined in Algorithm 4).*

R5 *Array slice: last n elements (a), first n elements (b), from indexes m to n-1 or from m^{th}-to-last to n^{th}-to-last (c)*
(a) **trans**(<JP:F>[-<start>:]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), -<start>)
(b) **trans**(<JP:F>[:<end>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <end>)
trans(<JP:F>[0:<end>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <end>)
(c) **trans**(<JP:F>[<m>:<n>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), <m>, (<n>-<m>))
trans(<JP:F>[-<m>:-<n>]<JP>, <cond>) →
trans(<JP:F>.*<JP>, <cond>) **SLICE**(dotNotation(<JP:F>), -<m>, (<m>-<n>))

R6 *Calculated array index, e.g. $.p[(@.length - 1)].q*

(a) **trans**(<JP1>[(<num_expr>)]<JP2>, <cond>) → **NOT_SUPPORTED**
if <JP1> contains a wildcard or a JavaScript filter expression

(b) **trans**(<JP:F >[(<num_expr>)], <cond>) →
AND(**EXISTS**(<JP:F >),
WHERE('this<JP:F>[**replaceAt**("this<JP:F >", <num_expr>)] **CONDJS**(<cond>')))

(c) **trans**(<JP1:F >[(<num_expr>)]<JP2:F>, <cond>) →
AND(**EXISTS**(<JP1:F >),
WHERE('this<JP1:F>[**replaceAt**("this<JP1:F >", <num_expr>)]<JP2:F>
CONDJS(<cond>')))

R7 *Heading wildcard*

(e) **trans**(.*<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))

(f) **trans**([*]<JP>, <cond>) → **ELEMMATCH**(**trans**(<JP>, <cond>))

R8 *Heading field name or array index*

(f) **trans**(.p<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)

(g) **trans**(["p"]<JP>, <cond>) → **FIELD**(p) **trans**(<JP>, <cond>)

(h) **trans**([i]<JP>, <cond>) → **FIELD**(i) **trans**(<JP>, <cond>)

R9 *No other rule matched, the current expression is not supported*
trans(<JP>, <cond>) → **NOT_SUPPORTED**

7.4.2.1 Rules R0 and R1

Rule R0 is the entry point of the translation process. The heading "$" character is simply removed and the process goes on with the rest of the expression.

Conversely, rule R1 is the termination point: when the JSONPath expression has been fully parsed, the last element that is created is the MongoDB condition, *i.e.* "`$exists:true, $ne:null`" for a *isNotNull* condition, and "`$eq:value`" for an *equals(value)* condition. If value is of a string datatype, it is surrounded with quotes.

7.4.2.2 Rule R2: Field Alternative

A field alternative or array index alternative is translated into an OR clause, corresponding to the MongoDB `$or` operator. The MongoDB `$or` operator cannot be used as a condition on a field; for instance, the following query is invalid: `"p.q": {$or: [{$eq: 10}, {$eq: 10}]}`.

Instead, the `$or` operator has to be either at the top-level query or nested in an `$elemMatch`, `$and` or `$or` operator. For this reason, a sequence of field names and/or array indexes (denoted by <JP:F>) must precede the alternative pattern (["p","q",...] or [i,j,...]). In the rewriting, the <JP:F> sequence is prepended to each member of the OR. In the example below, the ".p" stands for the <JP:F> term:

Condition

```
equals($.p.["q", "r"], 10)
```

is translated into

```
OR(FIELD(p) FIELD(p) COND(equals, 10), FIELD(p) FIELD(r) COND(equals, 10))
```

that, in turn, shall be translated into:

```
$or: [{"p.q": {$eq: 10}}, {"p.r": {$eq: 10}}]
```

Note that no assumption is made as to what may come after the alternative pattern, this is denoted in the rule by the JSONPath expression <JP> following the alternative pattern.

7.4.2.3 Rule R3: Heading Field Alternative

Rule R3 matches an expression with a heading field alternative or array index alternative. Contrary to rule R2, the alternative pattern is not preceded by a <JP:F> sequence. This case occurs when the alternative is either the first pattern in the JSONPath expression, or when it comes after a term such as a JavaScript filter (R4), an array slice (R5) or a wildcard (R7). Example:

Condition

```
equals($.p.*["q", "r"], 10)
```

is translated into:

```
"p": {$elemMatch: {$or: [{"q": {$eq: 10}}, {"r": {$eq: 10}}]}}
```

7.4.2.4 Rule R4: JavaScript Filter

A JavaScript (JS) filter is a boolean condition evaluated against elements of an array, where the "@" character stands for each array element, *e.g.* "`$.people[?(@.role)]`" matches all document elements of array "people" that have a field "role". Since a JS filter specifies a condition on all array elements, it is translated into a MongoDB query embedded in an `$elemMatch` operator. Function *transJS* (section 7.4.2.10) parses the JS expression and translates it. Example:

Condition

```
equals($.p[?(@.q)].r.*, "value")
```

is translated into:

```
"p": {$elemMatch: {
   "r": {$elemMatch: {$eq:"value"}},
   "q": {$exists:true}}}
```

R4 produces the first `$elemMatch` as well as the condition `"q":{$exists:true}`. The second `$elemMatch` is produced by rule R7 when processing the wildcard.

7.4.2.5 Rule R5: Array Slice

JSONPath and MongoDB query languages have two different ways of denoting array slices. JSONPath uses notation `[<start>:<end>:<step>]`, where all three terms are optional, and `<start>` and `<end>` may be negative. MongoDB uses two projection notations: `{$slice: <count>}` *or* `{$slice: [<start>, <count>]}`, where `<count>` may be negative in the first notation only, and `<start>` may be negative in the second notation. In JSONPath and MongoDB, a negative value indicates that the count starts from the end of the array. Due to these notation discrepancies, the rewriting of JSONPath slices into

MongoDB projections has limitations explicated in the table below, where n and m denote positive integers:

JSONPath	Semantics	MongoDB query language
array[0:n], array[:n]	First *n* elements: from index *0* to index *n-1*	"array" : {$slice: n}
array[-m:]	Last *m* elements	"array" : {$slice: -m}
array[m:]	From index *m* until the last element	n/a
array[m:n]	From index *m* to index n-1	"array" : {$slice: [m, (n-m)]}
array[-m:-n]	From m^{th} to last to $n\text{-}1^{th}$ to last	"array" : {$slice: [-m, (m-n)]}
array [m:n:s]	From index *m* to index *n-1* by step s	n/a

Consequently rules R5 (a), (b) and (c) do not cover all cases. Non-supported forms of JSONPath slice shall be treated in the default rule R9.

The JSONPath array slice notation is rewritten into the `$slice` operator that, unlike in other rules, is used as a projection parameter of the MongoDB *find()* method. Rule R5 must translate the JSONPath expression that comes before the array slice (<JP:F>) as well as the subsequent JSONPath expressions (<JP>) to generate the query parameter of the *find()* method. It does so by replacing the array slice by a wildcard ".*": `trans(<JP:F>.*<JP>, <cond>)`. Hence, the query part applies to the whole array, while the projection part shall select only the expected elements.

7.4.2.6 Rule R6: Calculated Array Index

A JSONPath calculated array index selects an element from an array using a JavaScript expression that evaluates to a positive integer. The script expression uses the "@" character instead of "this" to refer to the array.

Note: Rule 6 uses function *replaceAt*(<rep>, <path>) that replaces any occurrence of the '@' character with <rep> in string <path>. *E.g.:*

`replaceAt("this.people", "@ < 10")` returns `"this.people < 10"`.

Let us consider this condition: `equals($.staff[(@.length - 1)].name, "John");` it matches all documents in which the last element of array "staff" has a field "name" with value "John". In MongoDB, there is no way to retrieve the size of an array nor to calculate such an index (the $size operator is not relevant here as it specifies a condition on the size of an array but it cannot be used to obtain the array size). The only way to specify a condition on an element whose index is calculated is to use the `$where` operator. For instance, condition

```
equals($.staff[(@.length - 1)].name, "John")
```

shall be translated by rule R6(c) into:

```
$and:[{"staff":{$exists: true}}, {
       $where:"this.staff[this.staff.length - 1].name == 'John'"}
     ]
```

However, we have mentioned earlier that the `$where` operator can be used only as a member of the top-level query. Several situations may occur then:

- If a calculated array index is preceded by either a wildcard or a filter, its translation shall be embedded into an `$elemMatch` created by rules R7 or R4 respectively. Rule R6(a) makes this case impossible by returning NOT_SUPPORTED.
- Yet, rule R6 (b and c) produces a `$where` operator nested in an `$and` operator. We show in section 7.4.3 that we can rewrite a query containing a `$where` nested in a combination of `$and` and `$or` operators into a union of valid MongoDB queries in which `$where` operators show only in the top-level query.

If the calculated array index is followed by a non-empty JSONPath expression, that subsequent expression has to be part of the JavaScript expression in the `$where` operator. This is exemplified by the "name" field in the example above. Therefore, more generally, anything that follows the calculated array index should be rewritten into JavaScript. This is not always possible however, as illustrated by the two examples below:

(1) Condition `equals($.p[(@.length - 1)].*, "val")`, could be rewritten in:

```
$where:{"this.p[this.p.length-1].* == 'val'"}.
```

But this query is invalid since there is no equivalent to the wildcard in JavaScript.

(2) Similarly, condition `equals($.p[(@.length - 1)].r[?(@.q)].s, "val")` could be rewritten into:

```
$and: [{p:$exists}, {$where: "this.p[this.p.length - 1].r[?(@.q)].s == 'val'"}].
```

But again, this query is invalid since there is no simple JavaScript equivalent to the JSONPath notation "`?(@.q)`". Yet, in both situations, we could figure out a way to achieve the translation through the definition of a JavaScript function that would parse the array. At this stage however, we choose not to go down that road, and we further discuss this choice in section 8.6. Therefore, in rule R6(c) we restrict the type of terms that follow a calculated array index to a sequence of field names or array indexes, denoted by <JP2:F>.

7.4.2.7 Rule R7: Wildcard

As mentioned in section 7.3, the use of the wildcard within JSONPath expressions is restricted to the context of array fields, but it cannot apply to document fields. Hence, rule R7 simply translates a heading wildcard into an `$elemMatch` operator.

7.4.2.8 Rule R8: Field Name

Regular names and array indexes are translated into their equivalent dot-separated MongoDB path. Example: condition `isNotNull($.p[5]["s"])` is translated into "`"p.5.s": {$exists:true}`".

7.4.2.9 Rule R9: Non-Supported Cases

Rule R9 is the default rule. In case no other rule matched, the translation of the JSONPath expression to MongoDB query language is not supported. This applies in the following cases:

- A calculated array index is preceded or followed by a wildcard, an alternative or a JavaScript filter, as explained in rule R6.
- Unsupported array slice notation such as [m:n:s].
- JSONPath expressions entailing that the root document is an array field and not a document field, such as `$.*`, `$[1,2,…]`, `$[?(…)]` and `$[(…)]`.

7.4.2.10Translation of a JavaScript filter to MongoDB

Recursive function *transJS* translates a JavaScript filter into a MongoDB query. It consists of a set of rules, explicated in Algorithm 4, that apply if the JavaScript expression matches a certain pattern. The JavaScript expression is checked against the patterns in the order of the rules. When a match is found, the corresponding rule is applied and the search stops.

We use the following conventions:

- **<JSpath>**: denotes a non-empty JavaScript sequence of field names and array indexes, *e.g.* '.p.q.r', '.p[10]'.
- The **dotNotation(<JS_expr>)** function converts a JavaScript path to a MongoDB query path consisting of field names and array indexes in dot notation. It removes the optional heading dot. *e.g.* "dotNotation(.p[5]r)" returns "p.5.r".
- The **transJsOp(op)** function converts a JavaScript comparison operator into its MongoDB equivalent: `===` → `$eq`, `==` → `$eq`, `!=` → `$ne`, `<=` → `$lte`, `>=` → `$gte`, `<` → `$lt`, `>` → `$gt`, `=~` → `$regex`.

The expressiveness of the MongoDB query language, in terms of comparison, is quite limited compared to JavaScript Boolean conditions. As a result, when a JavaScript comparison cannot be turned in an equivalent MongoDB query, the rule returns the NOT_SUPPORTED clause that shall be used during the final translation phase.

Algorithm 4: Translation of a JavaScript filter into a MongoDB query (function transJS)

J0	**transJS**(<JS_expr1> && <JS_expr2>) → **AND**(**transJS**(<JS_expr1>), **transJS**(<JS_expr2>))
J1	**transJS**(<JS_expr1> || <JS_expr2>) → **OR**(**transJS**(<JS_expr1>), **transJS**(<JS_expr2>))
J2	**transJS**(@<JS_expr1> <op> @<JS_expr2>) → **NOT_SUPPORTED** *where <op> stands for one of {==, ===, !=, !==, <=, <, >=, >, %}*
J3	**transJS**(@<JSpath>) → **EXISTS**(**dotNotation**(<JSpath>))
J4	**transJS**(!@<JSpath>) → **NOT_EXISTS**(**dotNotation**(<JSpath>))
J5	(a) **transJS**(@<JSpath>.length == <i>) → **COMPARE**(**dotNotation**(<JSpath>), $size, <i>) (b) **transJS**(@<JSpath>.length <op> <i>) → **NOT_SUPPORTED** *where <op> stands for one of {!=, <=, <, >=, >, %}*
J6	**transJS**(@<JSpath> <op> <v>) → **COMPARE**(**dotNotation**(<JSpath>), **transJsOp**(<op>), <v>)
J7	**transJS**(<JS_expr>) → **NOT_SUPPORTED**

Rules J0 and J1 deal with the logical AND and OR JavaScript operators.

Rule J2 addresses the comparison of two document fields or two array fields such as "@.name != @.login". This is not permitted in MongoDB query language, yet it is possible to translate this condition using the `$where` operator. Typically, rule J2 could return:

AND(**EXISTS**(<JS_expr1>), **EXISTS**(<JS_expr2>), **WHERE**("this<JS_expr1> <op> this<JS_expr2>"))

Nonetheless, the *transJS* function is used only in the context of an `$elemMatch` (rule R4), and the `$where` operator is valid only in the top-level query. Therefore, rule J2 returns NOT_SUPPORTED.

Rules J3 and J4 deal with existential comparisons.

Rule J5 addresses tests on the length of an array field. The MongoDB `$size` operator allows for an equality test on the length of an array, but other types of comparison are not allowed. Similarly to the above discussion on rule J2, a `$where` operator could be used in J5(b) to return:

WHERE(this<JSpath>.length <op> <i>)

But again, the `$where` operator is valid only in the top-level query, and the *transJS* function is used only in the context of an `$elemMatch` (rule R4). Consequently, rule J5(b) returns NOT_SUPPORTED.

Rule J6 addresses all other types of supported comparison between a field and a literal value `<v>`.

Finally, rule J7 applies when no other rule matched. It is used as the default for all non-supported types of JavaScript expression.

7.4.3 Optimization and Translation into Concrete MongoDB Queries

Rules R0 to R9, defined in section 7.4.2, translate a condition on a JSONPath expression into an abstract representation of a MongoDB query. Yet, three potential issues hinder rewriting the abstract MongoDB query into a concrete MongoDB query:

(i) NOT_SUPPORTED clauses may indicate that a part of the JSONPath expression could not be translated into equivalent MongoDB operators.
(ii) A WHERE clause may be nested beneath a sequence of AND and/or OR clauses, although the MongoDB `$where` operator is valid only in the top-level query.
(iii) Unnecessary complexity such as nested ORs, nested ANDs, sibling WHEREs, etc., may produce an underperforming target query.

These issues are addressed by means of two additional sets of rewriting rules, O1 to O5 and W1 to W6, defined in sections 7.4.3.1 and 7.4.3.2 respectively. Function *rewrite* (section 7.4.3.3) repeatedly enforces these rules to perform all possible rewritings. It ends up with either one concrete MongoDB query or a union of concrete MongoDB queries. This whole rewriting process is depicted by step 2 in Figure 12.

7.4.3.1 Abstract MongoDB Query Optimization

Rules O1 to O5, pictured in Algorithm 5, cope with the above issues (i) and (iii). Each rule applies to queries that match the pattern in the head of the rule. More precisely:

- Rules O1 to O4 address issue (iii) by flattening nested OR, AND and UNION clauses, and merging sibling WHERE clauses.
- Rule O5 addresses issue (i) by removing NOT_SUPPORTED clauses. It ensures that the query returns a superset including at least all the correct answers. The transformation of these results into RDF triples shall produce all triples matching the SPARQL query, in addition to triples that may not match the query. In section 7.5 we propose an algorithm that orchestrates the overall SPARQL-to-MongoDB query processing: it completes the process with a *late SPARQL query evaluation* whose role is to rule out those additional triples that do not match the SPARQL query.

Algorithm 5: Optimization of an abstract representation of a MongoDB query.
The right arrow is to be read "is rewritten into".

O1 *Flatten nested AND, OR and UNION clauses:*
AND(C_1,… C_n, **AND**(D_1,… D_m,)) → **AND**(C_1,… C_n, D_1,… D_m)
OR(C_1,… C_n, **OR**(D_1,… D_m,)) → **OR**(C_1,… C_n, D_1,… D_m)
UNION(C_1,… C_n, **UNION**(D_1,… D_m,)) → **UNION**(C_1,… C_n, D_1,… D_m)

O2 *Merge ELEMMATCH with nested AND clauses:*
ELEMMATCH(C_1,… C_n, **AND**(D_1,… D_m,)) → **ELEMMATCH**(C_1,… C_n, D_1,… D_m).

O3 *Group sibling WHERE clauses:*
OR(..., **WHERE**("W1"), **WHERE**("W2")) → **OR**(..., **WHERE**("(W1) || (W2)")).
AND (..., **WHERE**("W1"), **WHERE**("W2")) → **AND**(..., **WHERE**("(W1) && (W2)")).
UNION(..., **WHERE**("W1"), **WHERE**("W2")) → **UNION**(..., **WHERE**("(W1) || (W2)")).

O4 *Replace AND, OR or UNION clauses of one term with the term itself.*
This situation may occur after flattening nested clauses or grouping sibling WHERE clauses.

O5 *Remove NOT_SUPPORTED clauses:*
(a) **AND**(C_1,… C_n, **NOT_SUPPORTED**) → **AND**(C_1,… C_n)
(b) **ELEMMATCH**(C_1,… C_n, **NOT_SUPPORTED**) → **ELEMMATCH**(C_1,… C_n)
(c) **OR**(C_1,… C_n, **NOT_SUPPORTED**) → **NOT_SUPPORTED**
(d) **UNION(**C_1,… C_n, **NOT_SUPPORTED**) → **NOT_SUPPORTED**
(e) **FIELD**(…)… **FIELD**(…) **NOT_SUPPORTED** → **NOT_SUPPORTED**

Below we further elaborate on the logics of rule O5. Let C_1,… C_n be any clauses and N be a NOT_SUPPORTED clause. Notation $C_1 \wedge \ldots \wedge C_n$ denotes the set of documents matching all conditions C_1 to C_n.

- O5(a): If a NOT_SUPPORTED clause occurs in an AND clause, it is simply removed. Since "$C_1 \wedge \ldots \wedge C_n \wedge N$" is more specific than "$C_1 \wedge \ldots \wedge C_n$" ($C_1 \wedge \ldots \wedge C_n \subseteq C_1 \wedge \ldots \wedge C_n \wedge N$), by simply removing the N component the whole condition is widened. Consequently, all matching documents shall be returned, but non-matching documents may be returned too, that shall be ruled out later on.
- O5(b): A logical AND implicitly applies to members of an ELEMMATCH clause. Therefore, removing the NOT_SUPPORTED has the same effect as in O5(a).
- O5(c) and O5(d): Contrary to the AND and ELEMMATCH cases, we cannot simply remove the NOT_SUPPORTED clause from an OR or UNION clause as the query would only return a subset of

the matching documents ($C_1 \vee \ldots \vee C_n \subseteq C_1 \vee \ldots \vee C_n \vee N$). Instead, the OR or UNION clause is replaced with a NOT_SUPPORTED clause. Consequently, the issue is raised up to the parent clause, and it shall be managed at the next rewriting iteration. Iteratively, we raise up the NOT_SUPPORTED clause until it is eventually removed (cases AND or ELEMMATCH above), or it ends up in the top-level query. The latter is the worst case in which the query is no longer selective at all and shall retrieve all the documents from the logical source.

- O5(e): Similarly to O5(c) and O5(d), a sequence of fields followed by a NOT_SUPPORTED clause is replaced with a NOT_SUPPORTED clause to raise the issue up to the parent clause.

Example. We illustrate Algorithm 5 with a dedicated example. We consider the purposely complicated condition below, that could be described in natural language as: "a member of team 0 is a beginner, has score 3, is either a player or a goal but not both, and his name is john".

```
equals($.teams.0[?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)].name, john")
```

Below we detail how the *trans* function translates this condition into an abstract MongoDB query, and we mention the rules applied at each step:

```
      trans($.teams.0[?(@.level=="beginner" && @.score>=3 &&
                        @.isPlayer<>@.isGoal)].name, equals("john")) =

R0,R8 FIELD(teams.0) trans([?(@.level=="beginner" && @.score>=3 &&
                              @.isPlayer<>@.isGoal)].name, equals("john"))

R4    FIELD(teams.0) ELEMMATCH( trans(.name, equals("john")),
          transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)]))

R8,R1 FIELD(teams.0) ELEMMATCH( FIELD(name) COND(equals, "john"),
         transJS([?(@.level=="beginner" && @.score>=3 && @.isPlayer<>@.isGoal)]))

J0,J6 FIELD(teams.0) ELEMMATCH(
                        FIELD(name) COND(equals, "john"),
                        AND( COMPARE(level, ==, "beginner"),
                             AND( COMPARE(@.score, >=, 3), NOT_SUPPORTED )
                        )
                     )
```

Notice that rule J6 translates condition `@.isPlayer<>@.isGoal` into a NOT_SUPPORTED clause since MongoDB cannot compare fields of a JSON document. From this stage, rule O1 flattens nested ANDs, and rule O2 removes the unnecessary AND clause beneath the ELEMMATCH:

```
O1    FIELD(teams.0) ELEMMATCH(
                        FIELD(name) COND(equals, "john"),
                        AND( COMPARE(level, ==, "beginner"),
                             COMPARE(score, >=, 3),
                             NOT_SUPPORTED))

O2    FIELD(teams.0) ELEMMATCH(
                        FIELD(name) COND(equals, "john"),
                        COMPARE(level, ==, "beginner"),
                        COMPARE(score, >=, 3),
                        NOT_SUPPORTED)
```

Lastly, rule O5 takes care of removing the NOT_SUPPORTED clause:

```
O5    FIELD(teams.0) ELEMMATCH(
                        FIELD(name) COND(equals, "john"),
                        COMPARE(level, ==, "beginner"),
                        COMPARE(score, >=, 3))
```

This abstract MongoDB query can now be rewritten into a concrete query:

```
"teams.0": {$elemMatch: {"name":{$eq:"john"}, "level":{$eq:"beginner"}, "score":{$gte:3}}}
```

Yet, we know that part of the condition was ignored: "`@.isPlayer<>@.isGoal`". Therefore, that query may return documents where field `isPlayer` equals field `isGoal`. This shall be taken care of by the late SPARQL query evaluation (section 7.5).

7.4.3.2 Pulling-up WHERE Clauses

By construction, rule R6 ensures that a WHERE clause cannot be nested into an ELEMMATCH clause, but may show in an AND or an OR clause. Furthermore, Algorithm 5 flattens nested OR and nested AND clauses, and merges sibling WHERE clauses. As a result, a WHERE clause may be either in the top-level query (in that case the query is executable) or it may appear in one of the following patterns, where "W" stands for a WHERE clause: `OR(…,W,…)`, `AND(…,W,…)`, `OR(…,AND(…,W,…),…)`, `AND(…,OR(…,W,…),…)`. When such patterns occur, rules W1 to W6 (Algorithm 6) address issue (ii) by "pulling up" WHERE clauses to the top-level query. Here is an insight into the method:

- The implicit semantics of the top-level query is to apply a logical AND between its members. Hence, we can replace a top-level `AND(C,W)` with its members denoted by `(C,W)`, this results in pulling up the W clause to the top-level query.
- Since `OR(C,W)` is not a valid MongoDB query, it is replaced with query `UNION(C,W)` which has the same semantics but is processed differently: `C` and `W` are evaluated separately against the database, then the query processing engine computes the union.
- `AND(C,OR(D,W))` is rewritten into `OR(AND(C,D), AND(C,W))`[92], and the `OR` is replaced with a `UNION`: `UNION(AND(C,D), AND(C,W))`. The W clause now shows in a top-level `AND` clause that can be replaced with its members. We finally get the query: `UNION((C,D), (C,W))`, where the W clause is at a top-level query.

Rewriting rules W1 to W4 (Algorithm 6) are a generalization of these examples. Since they may create UNION clauses nested beneath AND or OR clauses, additional rules W5 and W6 ensure that UNION clauses are pulled up to the top-level query. Note that rule O1 copes with the case of nested UNION clauses.

Algorithm 6: Pull-up of WHERE clauses to the top-level query.
The right arrow is to be read "is rewritten into".

W1	**OR**(C_1,...C_n, W) → **UNION**(**OR**(C_1,...C_n), W)
W2	**OR**(C_1,...C_n, **AND**(D_1,...D_m, W)) → **UNION**(**OR**(C_1,...C_n), **AND**(D_1,...D_m, W)) *Proof:* $C_1 \vee \ldots \vee C_n \vee (D_1 \wedge \ldots \wedge D_m \wedge W) \Leftrightarrow (C_1 \vee \ldots \vee C_n)$ **V** $(D_1 \wedge \ldots \wedge D_m \wedge W)$ Therefore, eval($C_1 \vee \ldots \vee C_n \vee (D_1 \wedge \ldots \wedge D_m \wedge W)$) = eval($C_1 \vee \ldots \vee C_n$) ∪ eval($D_1 \wedge \ldots \wedge D_m \wedge W$).

[92] This is a straightforward application of the theorem: $C \wedge (D \vee W) \Leftrightarrow (C \wedge D) \vee (C \wedge W)$.

W3 **AND**(C_1,...C_n, W) → (C_1,...C_n, W), *iif the AND clause is a top-level query object or under a UNION clause*.

W4 **AND**(C_1,...C_n, **OR**(D_1,...D_m, W)) → **UNION**(**AND**(C_1,...C_n, **OR**(D_1,...D_m)), **AND**(C_1,...C_n, W))
Proof: $C_1 \wedge \ldots \wedge C_n \wedge (D_1 \vee \ldots \vee D_m \vee W) \Leftrightarrow (C_1 \wedge \ldots \wedge C_n) \wedge ((D_1 \vee \ldots \vee D_m) \vee W)$
$\Leftrightarrow ((C_1 \wedge \ldots \wedge C_n) \wedge (D_1 \vee \ldots \vee D_m))$ **V** $((C_1 \wedge \ldots \wedge C_n) \wedge W)$
Therefore, eval($C_1 \wedge \ldots \wedge C_n \wedge (D_1 \vee \ldots \vee D_m \vee W)$) = eval($C_1 \wedge \ldots \wedge C_n \wedge (D_1 \vee \ldots \vee D_m)$) ∪ eval($C_1 \wedge \ldots \wedge C_n \wedge W$)

W5 **AND**(C_1,...C_n, **UNION**(D_1,...D_m)) → **UNION**(**AND**(C_1,...C_n, D_1),... **AND**(C_1,...C_n, D_m))
Proof: $C_1 \wedge \ldots \wedge C_n \wedge (D_1 \vee \ldots \vee D_m) \Leftrightarrow (C_1 \wedge \ldots \wedge C_n) \wedge (D_1 \vee \ldots \vee D_m)$
$\Leftrightarrow (C_1 \wedge \ldots \wedge C_n \wedge D_1) \vee \ldots \vee (C_1 \wedge \ldots \wedge C_n \wedge D_m)$
Therefore, eval($C_1 \wedge \ldots \wedge C_n \wedge (D_1 \vee \ldots \vee D_m)$) = eval($C_1 \wedge \ldots \wedge C_n \wedge D_1$) ∪… ∪ eval($C_1 \wedge \ldots \wedge C_n \wedge D_m$)

W6 **OR(**C_1,...C_n, **UNION**(D_1,...D_m)) → **UNION**(**OR**(C_1,...C_n), D_1, ...D_m))

Example. We illustrate rules W1 to W6 in a second dedicated example. The condition below states that the last member of either team “dev” or “test” has the name “john”:

```
trans($.teams["dev","test"][(@.length - 1)].name, equals("john"))
```

Function *trans* translates this condition into this abstract MongoDB query:

```
OR( AND(EXISTS(.teams.dev),
        WHERE('this.teams.dev[this.teams.dev.length - 1)].name CONDJS(equals("john"))')),
    AND(EXISTS(.teams.test),
        WHERE('this.teams.test[this.teams.test.length - 1)].name CONDJS(equals("john"))'))
)
```

Then, we iteratively apply rules O1 to O5 and W1 to W6 as described in function *rewrite* (next section). First, we apply subsequently rules W2 and O4 to replace the top-level OR with a UNION clause:

```
UNION(
    AND(EXISTS(.teams.dev),
        WHERE('this.teams.dev[this.teams.dev.length - 1)].name CONDJS(equals("john"))')),
    AND(EXISTS(.teams.test),
        WHERE('this.teams.test[this.teams.test.length - 1)].name CONDJS(equals("john"))'))
)
```

Now, the abstract MongoDB query is a union of two top-level AND operators that can simply be removed by rule W3:

```
UNION(
     (EXISTS(.teams.dev),
      WHERE('this.teams.dev[this.teams.dev.length - 1)].name CONDJS(equals("john"))')),
     (EXISTS(.teams.test),
      WHERE('this.teams.test[this.teams.test.length - 1)].name CONDJS(equals("john"))'))
)
```

Finally, we end up with a union of two concrete queries. Both queries can be evaluated against the database, ultimately the query processing engine shall union their results.

```
UNION( ( "teams.dev": {$exists: true},
         $where: 'this.teams.dev[this.teams.dev.length - 1)].name == "john"'),
       ( "teams.test": {$exists: true},
         $where: 'this.teams.test[this.teams.test.length - 1)].name == "john"')
     )
```

7.4.3.3 Rewritability and Completeness Properties

We define in Algorithm 7 the optimization and translation algorithm that repeatedly enforces rules O1 to O5 and W1 to W6 to perform all possible rewritings, so as to obtain a valid query, "valid" meaning that there is no more NOT_SUPPORTED clause, any WHERE clause only appears as a top-level query object, and a UNION clause may appear only as the top-level clause. Ultimately, the result can be translated into a concrete MongoDB query or a union of concrete MongoDB queries, as per Definition 16. Furthermore, this rewriting returns (at least) all matching documents. This result is formalized in Theorem 1 that we demonstrate hereafter.

Algorithm 7: Abstract MongoDB query optimization and translation into concrete MongoDB queries

Function rewrite(Q):
do
  do
    Q ← apply rules O1 to O5 that match any sub-query of Q
  until no more rewriting can be performed
  do
    Q ← apply rules W1 to W6 that match any sub-query of Q
  until no more rewriting can be performed
until no more rewriting can be performed by either rules O1 to O5 or W1 to W6
return Q

Theorem 1*. Let C be an equality or non-null condition on a JSONPath expression. Let Q = (Q_1, …Q_n) be the abstract representation of the MongoDB query produced by trans(C).*

Rewritability*: It is always possible to rewrite Q into a query Q' = UNION(Q'_1,… Q'_m) such that ∀i ∈[1, m] Q'_i is a valid MongoDB query, i.e. Q'_i does not contain any NOT_SUPPORTED clause, and a WHERE clause only shows at the top-level of Q'_i.*

Completeness*: Q' retrieves all the certain answers, i.e. all the documents matching condition C. If Q contains at least one NOT_SUPPORTED clause, then Q' may retrieve additional documents that do not match condition C.*

Proof of Theorem 1.

Completeness. The completeness property is a result of how rule O5 (described in details in section 7.4.3.1) deals with NOT_SUPPORTED clauses. When a NOT_SUPPORTED clause shows in an AND or ELEMMATCH clause, it is removed, thus widening the condition (O5 (a) and (b)). An OR or UNION clause containing a NOT_SUPPORTED clause is replaced with a NOT_SUPPORTED clause (O5 (c) and (d)), thus raising the issue up to the parent clause. Similarly, a sequence of FIELDS followed by a NOT_SUPPORTED clause is replaced with a NOT_SUPPORTED clause (O5(e)). Algorithm 7 repeatedly applies this rule until all NOT_SUPPORTED are eventually removed. The worst-case scenario occurs when a NOT_SUPPORTED clause ends up in the top-level query: the query is no longer selective at all. Nevertheless, this process ensures that all documents matching the condition are ultimately retrieved, possibly in addition to non-matching documents.

Rewritability, case of NOT_SUPPORTED clauses. By construction, function *trans* may generate a NOT_SUPPORTED clause in the top-level query or in the following patterns: AND(…,N,…), ELEMMATCH(…,N,…), OR(…,N,…), UNION(…,N,…), FIELD(…)…FIELD(…) N, where "N" stands for a NOT_SUPPORTED clause. If it is in the top-level query, then Definition 16 rewrites it into the empty query that shall retrieve all documents of the collection. In the case of other patterns, when applying rewriting rule O5 we obtain:

AND(…,N,…) → AND(…)
ELEMMATCH(…,N,…) → ELEMMATCH(…)
OR(…,N,…) → N
UNION(…,N,…) → N
FIELD(…)…FIELD(…) N → N

The first two rewritings remove the NOT_SUPPORTED clause, coming up with a valid query. The next three rewritings raise the NOT_SUPPORTED up to the parent clause. Since nested AND/OR/UNION clauses are merged by rule O1, this may lead to one of the patterns below and their subsequent rewritings:

AND(…,OR(…,N,…),…) → AND(…,N,…) → AND(…)
AND(…,UNION(…,N,…),…) → AND(…,N,…) → AND(…)
AND(…,FIELD(…)…FIELD(…) N,…) → AND(…,N,…) → AND(…)
ELEMMATCH(…,OR(…,N,…),…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)
ELEMMATCH(…,UNION(…,N,…),…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)
ELEMMATCH(…,FIELD(…)…FIELD(…) N,…) → ELEMMATCH(…,N,…) → ELEMMATCH(…)

The above rewritings show that, wherever the NOT_SUPPORTED clause shows, it is iteratively removed by rewriting rules O1 to O5 and W1 to W6. Hence the first part of the **rewritability** property: it is always possible to come up with a rewriting that does not contain any NOT_SUPPORTED clause.

Rewritability, case of WHERE clauses. By construction, function *trans* may generate a WHERE clause in the top-level query or nested within AND or OR clauses, but not within an ELEMMATCH clause. If multiple sibling WHERE clauses are generated, they are merged by rule O3. Furthermore, rules W1 to W6 may create UNION clauses, and rule O1 flattens nested OR/AND/UNION clauses. Consequently, a

WHERE clause may be either in the top-level query (the query is thus executable) or in the following patterns, where "W" stands for a WHERE clause:

OR(…,W,…)
OR(…,AND(…,W,…),…)
OR(…,UNION(…,W,…),…)
AND(…,W,…)
AND(…,OR(…,W,…),…)
AND(…,UNION(…,W,…),…)
UNION(…,W,…)
UNION(…,AND(…,W,…),…)
UNION(…,OR(…,W,…),…)

To prove Theorem 1, we must show that Algorithm 7 can rewrite a query so that the depth of a WHERE clause be 0. Toward that end, we define function *depth* that measures the depth of a MongoDB query consisting of AND, OR, UNION and WHERE clauses. First, we postulate:

depth(C_1/…/C_n) = depth(C_1) + … + depth(C_n)
where C_1/…/C_n are clauses nested within one another
depth(UNION) = 0
depth(AND) = 1
depth(OR) = 1

AND and OR clauses account for 1, but UNION accounts for 0: indeed UNION is not a MongoDB operator, instead it is meant to be processed outside of the database. Notation "C_1/…/C_n" represents a nested query in which clause C_1 is parent of clause C_2, C_2 is parent of clause C_3, etc. until clause C_n.

We define function *$depth_w(Q)$* as the depth of a clause WHERE within a query Q (assuming that Q contains exactly one WHERE clause):

$depth_w(C_1, \dots C_n, W) = 0$ *(case of a top-level query)*

$depth_w(C_1(\dots C_2(\dots C_n(\dots W)))) = depth(C_1/C_2/\dots/C_n)$

Below we explore how rules W1 to W6 rewrite the above listed patterns. For each one, we give the depth of the WHERE clause in the pattern and in the rewritten query.

OR(…,W,…)	*Rule W1:* Q: **OR**(C_1,...C_n, W) → Q': **UNION**(OR(C_1,...C_n), W) $depth_w$(Q) = 1 $depth_w$(Q') = 0
OR(…,**AND**(…,W,…),…)	*Rule W2:* Q: **OR**(C_1,...C_n, **AND**(D_1,...D_m, W)) → Q': **UNION**(OR(C_1,...C_n), **AND**(D_1,...D_m, W)) $depth_w$(Q) = 2 $depth_w$(Q') = 1
AND(…,W,…)	*Rule W3 (iif the AND clause is a top-level query or under a UNION clause):* Q: **AND**(C_1,...C_n, W) → Q': (C_1,...C_n, W) $depth_w$(Q) = 1 $depth_w$(Q') = 0

AND(…,**OR**(…,W,…),…)	*Rule W4:* Q: **AND**(C_1,...C_n, **OR**(D_1,...D_m, W)) → Q': **UNION**(**AND**(C_1,...C_n, **OR**(D_1,...D_m)), **AND**(C_1,...C_n, W)) $depth_w(Q) = 2$ $depth_w(Q') = 1$
AND(…,**UNION**(…,W,…),…)	*Rule W5:* Q: **AND**(C_1,...C_n, **UNION**(D_1,...D_m, W)) → Q': **UNION**(**AND**(C_1,...C_n, D_1),... **AND**(C_1,...C_n, D_m), **AND**(C_1,...C_n, W)) $depth_w(Q) = 1$ $depth_w(Q') = 1$
OR(…,**UNION**(…,W,…),…)	*Rule W6:* Q: **OR**(C_1,...C_n, **UNION**(D_1,...D_m, W) → Q': **UNION**(**OR**(C_1,...C_n), D_1, ...D_m, W)) $depth_w(Q) = 1$ $depth_w(Q') = 0$
UNION(…,W,…)	The WHERE clause is a top-level query, the query is valid as is and no rewriting is needed.
UNION(…,**AND**(…,W,…),…)	*Rule W3 (iif the AND clause is a top-level query or under a UNION clause):* Q: **UNION**(C_1,...C_n, AND(D_1,...D_m, W)) → Q': **UNION**(C_1,...C_n, (D_1,...D_m, W)) $depth_w(Q) = 1$ $depth_w(Q') = 0$
UNION(…,**OR**(…,W,…),…)	*We first apply rule W1 then rule O1 to merge nested UNIONs:* Q: **UNION**(C_1,...C_n**, OR**(D_1,...D_m, W)) → **UNION**(C_1,...C_n, **UNION**(OR(D_1,...D_m), W)) → Q': **UNION**(C_1,...C_n, OR(D_1,...D_m), W) $depth_w(Q) = 1$ $depth_w(Q') = 0$

In all the above patterns, the depth of the WHERE clause is always decreased by one using rules W1 to W6 and optionally rule O1, except for one where the depth is constant: in pattern **AND**(…,**UNION**(…,W,…),…), the resulting Q' query is:

UNION(**AND**(C_1,...C_n, D_1),... **AND**(C_1,...C_n, D_m), **AND**(C_1,...C_n, W))

Thus, $depth_w(Q) = depth_w(Q') = 1$. Nevertheless, a UNION is either a top-level query, and in that case the inner ANDs are replaced by their members, or the UNION is nested within some other query and it will eventually be raised up to the top-level query by rules W5 or W6. Hence, in all cases, we shall be able to come up with a query Q'' where $depth_w(Q'') = 0$.

By applying this process iteratively, it is easy to see that we ultimately come up with a rewriting that contains WHERE clauses only in the top-level query, hence the second part of the **rewritability** property.

7.5 Complete Query Translation and Evaluation Algorithm

Let us sum up the whole translation process so far, depicted in Figure 13. In step 1, function $trans_m$ (Chapter 6) translates a SPARQL graph pattern into an abstract query under a set of xR2RML mappings denoted by *m*. It leverages function $transTP_m$ to translate a triple pattern *tp* into an abstract query under the set of mappings bound to *tp* by function $bind_m$. The resulting abstract query contains atomic abstract queries of the form *{From, Project, Where, Limit}*. The *Where* part consists of *isNotNull*, *equals* and *sparqlFilter* conditions. In step 2, function *proj* (section 7.4.1) translates each projected JSONPath expression into a MongoDB projection argument, function *trans* (section 7.4.2) translates each *isNotNull* and *equals* condition on a JSONPath expression into an abstract representation of a MongoDB query, and function *rewrite* (section 7.4.3) optimizes and rewrites this abstract representation into a union of concrete MongoDB queries.

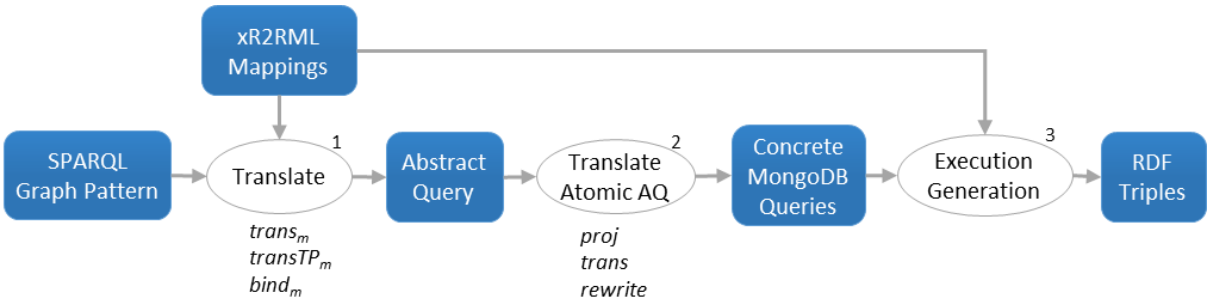


Figure 13: Complete SPARQL-to-MongoDB Query Translation and Evaluation

The last step that we have not mentioned yet is the execution of concrete MongoDB queries and the translation of results into actual RDF triples, depicted by step 3. To propose a complete approach, Algorithm 8 describes how the different steps are orchestrated, from the rewriting of SPARQL graph pattern until the final generation of the RDF triples that match it.

- Lines 2 and 3 address the translation of the SPARQL graph pattern into an optimized abstract query (step 1 in Figure 13).
- The for-loop from line 5 to line 25 deals with the individual atomic abstract queries. For each atomic query, the JSONPath expressions of the *Project* part are translated into a projection argument (lines 7-10), and the conditions of the *Where* part are translated into an abstract MongoDB query (lines 12-17) which is optimized and rewritten into a union of concrete MongoDB queries (line 18). Each concrete MongoDB query is then executed against the database and its results stored in set R_i (lines 20-24).
- On each R_i, the triples maps bound to Q_i are applied (line 27): this generates the RDF terms that the query processing engine can now use to compute the abstract query operators (line 28). This produces a primary result RDF graph.

Algorithm 8: Overall SPARQL-to-MongoDB query processing

```
1  Function process(sparqlGraphPattern):
2  abstractQuery ← trans_m(sparqlGraphPattern)
3  abstractQuery ← optimize abstractQuery // filter optimization and pushing,
4                  self-join and self-union elimination, constant projection, filter propagation
5  for each atomic abstract query Q_i = {From, Project, Where, Limit} ∈ abstractQuery do
6     // Translate projections of the Project part into a MongoDB projection argument
7     P_i ← ∅
8     for each projection ∈ Project do
9        P_i ← P_i, proj(projection)
10    end for
11    // Translate conditions of the Where part into an abstract MongoDB query
12    Q ← true
13    for each cond ∈ Where condition do
14       if cond is isNotNull(<JSONPath>), <condition> = isNotNull endif
15       if cond is equals(<JSONPath>, value), <condition> = equals(value) endif
16       Q ← AND(Q, trans(<JSONPath>, <condition>))
17    end for
18    Q_i' ← rewrite(Q)  // Q_i' is either a concrete query or a union of concrete queries
19    // Compute R_i, the set of documents matching Q_i'
20    if Q_i' is a concrete MongoDB query
21       R_i ← execute(Q_i', P_i, Limit)
22    else // Q_i' is UNION(q_1, ..., q_n)
23       R_i ← execute(q_1, P_i, Limit) ∪ ... ∪ execute(q_n, P_i, Limit)
24    end if
25 end for
26 // Generate the RDF triples corresponding to the SPARQL graph pattern
27 Apply triples maps bound to each Q_i to all documents of R_i
28 primaryGraph ← compute UNION, INNER JOIN, LEFT OUTER JOIN, FILTER and LIMIT operators
29 // Late SPARQL query evaluation
30 resultGraph ← evaluate sparqlGraphPattern against primaryGraph
31 return resultGraph
```

At this point, we cannot guarantee that the RDF graph we have produced contains only RDF triples that match the SPARQL graph pattern. Let us remind why:

(i) Some JSONPath elements are not supported in the rewriting process (see restrictions in section 7.3 and 7.4.2). Yet, we proved in section 7.4.3.3 that the rewriting shall retrieve all documents matching the SPARQL graph pattern, but additional non-matching documents may be returned too.

(ii) At this stage, our method does not rewrite SPARQL filters, embedded in atomic abstract queries using *sparqlFilter* conditions, into appropriate MongoDB operators.

Furthermore, the algorithm has produced a set of RDF triples, which is appropriate for a CONSTRUCT or DESCRIBE SPARQL query, but SELECT and ASK SPARQL queries expect a SPARQL Query Results, not an RDF triples.

To work out those issues, Algorithm 8 introduces a final step called the *late SPARQL query evaluation*: the initial SPARQL query is evaluated against the primary result RDF graph (line 30), which rules out all the non-matching triples that were generated due to any of the issues listed above.

7.6 Discussion and Perspectives

MongoDB find vs. aggregate queries. In a recent work, Botoeva et al. have proposed a generalization of the OBDA principles to support MongoDB [Botoeva et al., 2016a]. Their approach has similarities and discrepancies with ours, that we outline below.

Botoeva et al. derive a set of type constraints (literal, object, array) from the mapping assertions, called the MongoDB database *schema*. Then, a relational view over the database is defined with respect to that schema, notably by flattening array fields. A SPARQL query is rewritten into relational algebra (RA) query, and RA expressions over the relational view are translated into MongoDB *aggregate* queries. Similarly, we translate a SPARQL query into an abstract representation (that is not the relational algebra) under xR2RML mappings. To deal with the tree form of JSON documents we use JSONPath expressions. On the one hand, this avoids the definition of a relational view over the database, but this comes with additional complexity in the translation process, as translating conditions on JSONPath expressions is not straightforward. On the other hand, the advantage of our method is that the query evaluation relies on existing database indexes, whereas in the case of Botoeva et al, the flattening step prevents from exploiting the indexes.

The mappings are quite similar in both approaches although xR2RML is more flexible: (i) class names (in triples `?x rdf:type A`) and predicates can be built from database values whereas they are constant in the approach of Botoeva et al., and (ii) xR2RML allows to turn an array field into an RDF collection or container, while their work only supports the multiple-triples strategy.

Finally, Botoeva et al. produce MongoDB *aggregate* queries: the major advantage is that a SPARQL 1.0 query is translated into a single semantics-preserving target query, thus delegating the whole processing to MongoDB. Yet, in practice, *aggregate* queries model processing pipelines. In some cases, they may perform extremely poorly in terms of memory and CPU consumption. This issue has been identified by the authors. Hence, they suggest that it might be necessary to decompose RA queries into smaller sub-queries, and finally perform the remaining steps in the query-processing engine. Our approach produces *find* queries that are indeed less expressive, but whose performance is easier to anticipate. This puts a higher burden on the query-processing engine (joins, unions and filtering), but having the job done outside of the database engine allows to leverage extensive works about query plan optimizations [Haas et al., 1997; Schwarte et al., 2011; Görlitz & Staab, 2011; Macina et al., 2016], whereas this is not possible when the database performs an *aggregate* query in a black-box manner.

In the future, it would be interesting to see whether we could characterize mappings with respect to the type of query that shall perform best: single vs. multiple separate queries, *find* vs. *aggregate*, and figure out a balance between the two approaches.

Limitations. SPARQL filters are not tackled in the translation of an abstract query into the MongoDB query language. We plan to address this in the future, although it is likely that the support shall be limited by the capabilities of the underlying database. For instance, SQL supports most of the SPARQL operators such as logics, comparison, arithmetic and unary operators. This is far from being the case in MongoDB. JavaScript functions can help in this matter, although we have to consider this option with reluctance due to the performance issues it entails (discussed below). Again, some filtering tasks shall be delegated to the query-processing engine to bridge the gap between SPARQL and MongoDB.

Dealing with the MongoDB $where operator. In MongoDB *find* queries, the `$where` operator is valid only in the top-level query document. Using rewriting rules W1 to W6 (section 7.4.3.2), we showed that we can pull up a `$where` operator nested beneath AND or OR operators, but we cannot deal with a `$where` operator nested beneath an `$elemMatch` operator. By construction, rules in function *trans* (section 7.4.2) exclude the latter case by generating a NOT_SUPPORTED clause. In other words, *trans* drops the `$where` operator and postpones the evaluation of the condition to a later step: the effect is to widen the query that may retrieve more documents than those matching the initial SPARQL graph pattern. Then, in Algorithm 8, we run a late evaluation of the SPARQL query against the set of generated triples to make sure we produce only the expected triples.

An alternative is to push whatever needs to be done by the `$elemMatch` operator into a JavaScript function called by the `$where` operator. Let us consider the following example: a MongoDB instance stores JSON documents about bank account details, such as:

```
{accounts: [
   {current: { credits: 100, debits: 50}},
   {savings: { credits: 80, debits: 80}}
]}
```

We want to retrieve documents where credits equal debits in at least one account. The MongoDB `$eq` operator does not allow to specify the equality between two fields, therefore we must use the `$where` operator. We cannot write the following query: `{"accounts": {$elemMatch: {$where: {"credits == debits"}}}}` as the `$where` operator must be in the top-level query document. But we can write a JavaScript function that browses the "accounts" array to check if the condition is true for at least one element in the array:

```
$where: {function() {
  result = false;
  for (i = 0; i < this.accounts.length; i++)
    result = result || ( this.accounts[i].credits == this.accounts[i].debits);
  return result }}
```

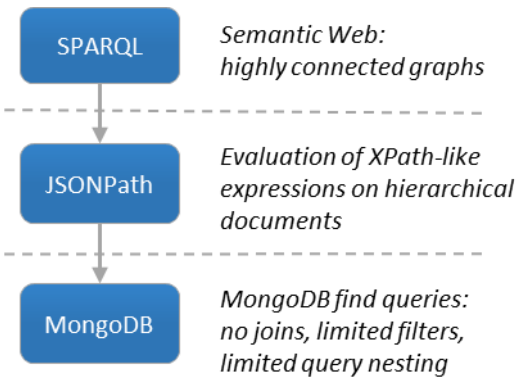
This option has the advantage of returning only the matching documents, but it has two shortcomings: (i) it may cause a serious performance penalty in the database: as we already mentioned, MongoDB cannot take advantage of indexes when executing JavaScript code, thus it shall retrieve all documents

matching all conditions except the `$where`, then apply the JavaScript function to all of them; (ii) it can lead to the generation of complex JavaScript functions when it comes to translate richer JSONPath expressions. Conversely, in the method we have chosen, the database query shall perform faster but the price is a larger amount of data retrieved and an additional SPARQL query evaluation to rule out non-matching triples. It is unclear, at this stage, whether one solution should be preferred to the other. But most likely, we can assume that the choice shall depend on the context.

7.7 Conclusion

In Chapter 6, we have exhibited a method aimed at fostering the development of SPARQL interfaces to heterogeneous databases. Toward that end, and to avoid defining yet another SPARQL translation method for each and every target database, we adopted a two-phase approach. The first phase covers all the database-independent steps: a SPARQL query is translated into an optimized abstract query, utilizing xR2RML to describe the mapping of a target database to an RDF representation that may rely on existing domain ontologies. The second phase enacts the steps that are specifically dependent on the target database, by translating the abstract query into the target database query language.

To demonstrate the effectiveness of the method, in Chapter 7 we have considered the case of a popular NoSQL document store, MongoDB. Thereby, we have enabled the SPARQL access to arbitrary MongoDB documents. However, we have shown that translating an abstract query into MongoDB *find* queries is a challenging step. The challenge arises from the discrepancy between the expressiveness of the different models that we traverse during the rewriting process.



(1) SPARQL comes from the Semantic Web world; it is tailored to the querying of highly connected graphs, and supports rich filtering capabilities inherited from XPath functions.

(2) Since MongoDB stores JSON documents, in the xR2RML mappings we use the JSONPath language as a syntactic bridge between RDF and MongoDB documents. JSONPath is used to extract data elements from MongoDB documents. Later on, these data elements are translated into RDF terms. Consequently, at the abstract query level, we translate SPARQL triple patterns into conditions on JSONPath expressions.

(3) Finally, we reach the level of the MongoDB query language: it is meant to query isolated documents where data redundancy is the norm and cross-references are the exception, while ensuring high throughput on large distributed infrastructures. As a result, its expressiveness is much more limited than SPARQL and even JSONPath: joins are not supported, filters are supported with strong restrictions (the comparison between two fields of a document is not allowed, hardly any string manipulation function), and the nesting of queries within one another is hardly supported. Ultimately, the best we can do is to translate conditions on JSONPath expressions into MongoDB queries, while leaving the processing of higher-level operators (joins, unions, SPARQL

filters) to the query-processing engine. Yet, not all JSONPath expressions can be fully translated into equivalent MongoDB operators.

Consequently, the query translation method cannot ensure that query semantics be preserved. Even so, we proved that rewritten queries retrieve all matching documents, in addition to possibly non-matching ones, and we have worked out this issue by evaluating the SPARQL query against the RDF triples generated from the database results. This guarantees semantics preservation, at the cost of an additional SPARQL evaluation.

Chapter 8. Experimentation and Evaluation: Use Case in Digital Humanities

8.1 Introduction

In the previous chapters, we have first defined the xR2RML generalized mapping language, meant to enable the mapping of heterogeneous databases to an RDF representation that may align the data on existing vocabularies and ontologies. Secondly, we have formalized a method to translate a SPARQL query into a pivot abstract query by matching the SPARQL graph pattern with xR2RML mappings. This step is independent of any target database and enables the querying of data in various formats. Finally, to illustrate the interest of the abstract query language, we have chosen the MongoDB NoSQL document store as an example database, and we devised a method to translate an abstract query into MongoDB queries.

This chapter illustrates the effectiveness of the whole approach in the context of a real-world use case, thus covering: (i) the ability of xR2RML to map an existing MongoDB database towards a non-trivial RDF representation, and (ii) the ability to query a MongoDB database using SPARQL, based on these xR2RML mappings. To date, to our knowledge, the only other approach meant to rewrite from SPARQL to MongoDB is the *ontop* software initially designed for relational databases [Botoeva et al., 2016a]. Unfortunately, the MongoDB-enabled version of ontop is not yet available for tests. As a result, we could not compare the performances of our implementation with any other product. Besides, there does not exist any benchmark for querying MongoDB databases in SPARQL, like the Berlin SPARQL Benchmark for relational databases [Bizer & Schultz, 2009]. Therefore, in this chapter we provide various performance measures, either in the graph materialization or the query rewriting access modes, but we cannot compare those measures with any other approach at this time.

The outline of this chapter is as follows. We first briefly describe the Morph-xR2RML prototype implementation (section 8.2). Then, we introduce the context in which our experimentation is conducted: we describe TAXREF, a taxonomical reference used in conservation biology, along with the SKOS thesaurus that we derived thereof (section 8.3). In section 8.4, we use xR2RML to map the MongoDB database hosting TAXREF into the chosen SKOS representation, and give performance elements with respect to the graph materialization. In section 8.5, we measure the SPARQL-to-MongoDB performances using various example queries. In particular, we measure the gain of optimizing abstract queries, keeping in mind that these optimizations are independent of the target database.

Notations. The examples of this section use the namespace definitions below.

Prefix	IRI
rr	http://www.w3.org/ns/r2rml#
xrr	http://www.i3s.unice.fr /ns/xr2rml#
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
skos	<http://www.w3.org/2004/02/skos/core#>
skosxl	<http://www.w3.org/2008/05/skos-xl#>
dct	<http://purl.org/dc/elements/1.1/>
txn	<http://lod.taxonconcept.org/ontology/txn.owl#>
gn	<http://www.geonames.org/ontology#>
nt	<http://purl.obolibrary.org/obo/ncbitaxon#>
dwc	<http://rs.tdwg.org/dwc/terms/>
taxref	<http://inpn.mnhn.fr/taxref/9.0/>
taxrefrk	<http://inpn.mnhn.fr/taxref/taxrank#>
taxrefhab	<http://inpn.mnhn.fr/taxref/habitat#>
taxrefbgs	<http://inpn.mnhn.fr/taxref/bioGeoStatus#>
taxrefprop	<http://inpn.mnhn.fr/taxref/properties/>

8.2 The Morph-xR2RML Prototype Implementation

To evaluate the effectiveness of the xR2RML mapping language, we have developed a prototype implementation, *Morph-xR2RML*, that comes with connectors for the MySQL and Postgres relational databases, and for the MongoDB document store. It can process a database in either the data materialization or the query rewriting modes. In query rewriting mode, it can run as a fully compliant SPARQL 1.1 endpoint. It is extensible by design: supporting a new type of database consists in the creation of a software module that provides implementation for a set of interfaces, thus encapsulating and isolating any database-specific concerns from the rest of the project code.

Morph-xR2RML is available on GitHub[93] under the Apache 2.0 license, it is written in the Scala programming language, and based on Morph-RDB [Priyatna et al., 2014], an implementation of R2RML that we have extended to support xR2RML specificities. We performed a substantial code refactoring in order to isolate any RDB-related code into a dedicated software module, leaving all other common functions in database-agnostic modules. We extended the existing code to support xR2RML features with relational databases, and we developed a connector to the MongoDB document store, to translate MongoDB JSON documents into RDF and rewrite SPARQL queries into MongoDB queries.

Software architecture. Figure 14 depicts the software modules that Morph-xR2RML consists of; dependencies are read top to down, i.e. the project on the top is the root on which all others depend. Modules *morph-core*, *morph-xr2rml-lang* and *morph-base* are all database-independent. All database-specific issues are addressed in the *morph-xr2rml-mongo* and *morph-xr2rml-rdb* modules. We briefly describe each of them:

[93] xR2RML GitHub repository: https://github.com/frmichel/morph-xr2rml/

- The *morph-core* module brings major global definitions: constants, various utility classes (properties, exceptions, RDF/JSON/XML manipulation and serialization) and mixed syntax path utilities.
- The *morph-base* module provides abstractions for major functions of the xR2RML processing engine: accessing and reading an abstract database, translating data elements into RDF terms, materializing RDF triples, translating from SPARQL to the Abstract Query Language, etc.
- The *morph-xr2rml-lang* module contains the object model representing the xR2RML mapping language elements.
- The *morph-xr2rml-mongo* module implements the connector to a MongoDB database, the graph materialization and the query rewriting from the Abstract Query Language to the MongoDB query language.
- The *morph-xr2rml-rdb* module implements the materialization and query rewriting engine for SQL databases, it is mostly the legacy code of Morph-RDB.
- Finally, the *morph-xr2rml-dist* module generates a distributable archive including the MongoDB and RDB engines into a single JAR file, along with the program entry point (the main class), the SPARQL endpoint service code, as well as example databases, mappings and engine configuration files.

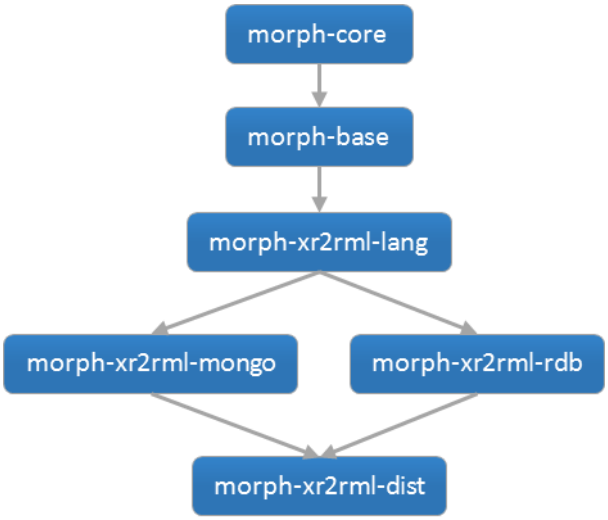


Figure 14: The Morph-xR2RML software architecture. The arrows is a parent-child module relation

Functional tests. We conducted an early evaluation phase focused on (i) the non-regression with respect to R2RML features, and (ii) the validation of new xR2RML features. We ran tests against two simple databases: a MySQL relational database and a MongoDB database with two collections. In both cases, the data and associated xR2RML mappings were designed to cover most mapping situations addressed by xR2RML: strategies for handling multiple RDF terms, JSONPath and XPath expressions, mixed-syntax paths for content with mixed formats (relational, JSON, XML, CSV/TSV), cross-references, production of RDF collection/containers, management of UTF-8 characters. A dump of both databases as well as the example mappings are available on the GitHub repository.

Limitations. At the time of writing, the prototype has the following limitations:

(i) The generation of RDF collections and containers is supported in all cases (e.g. multiple values resulting of the evaluation of a data element reference or a mixed-syntax path, result of a join

query implied by a referencing object map), except in the case of a regular R2RML join query applied to a relational database: the result of an SQL join query cannot be translated into an RDF collection or container.

(ii) Only simple `NestedTermMaps` are implemented *i.e.* to qualify RDF terms generated within an RDF collection/container. More complex nested term maps (with recursive parsing using another nested term map and using `xrr:reference` or `rr:template` properties) are not supported.

(iii) Named target graphs are not supported.

8.3 Construction of a SKOS Zoological and Botanical Reference Thesaurus

The Zoomathia research network[94] aims at studying the transmission of zoological knowledge throughout the Middle Age and Late Antiquity, by integrating heterogeneous data sources. More precisely, it intends to leverage the Semantic Web technologies to annotate and link together rich mediaeval compilation literature on Ancient zoological knowledge, archaeozoological data from excavation reports, iconographic material and modern conservation biology knowledge.

An increasing amount of historical heritage material is encoded in domain-specific digital formats. For instance, the SourcEncyMe[95] and Ichtya[96] projects aim to encode mediaeval encyclopedias in the XML-TEI standard[97] while adding manual annotations with regards to mediaeval compilers, alleged author sources and referenced taxa. The Biblissima project[98] is an observatory for medieval and renaissance written cultural heritage [Bonicel, 2013]. It federates over 40 research projects that contribute material in the form of digitized ancient manuscripts alongside metadata. One of its outcomes is the Biblissima ontology[99] that leverages existing library and museum organization systems to allow study the historical transmission of the manuscripts. These works succeed in making it easier to discover and exploit scientific material within the scientific community, but sharing those resources with other scientific communities remains hampered by the lack of formal semantic reference and terminological standards. For instance, the dolphin is a research topic for modern studies on biodiversity, for archaeozoologists, as well as for studies on Greek mythology wherein the dolphin played an important symbolic role. Nevertheless, when the dolphin is identified in the TEI annotation of the *Hortus Sanitatis* mediaeval encyclopedia[100] or in Pliny the Elder's work, *Naturalis Historia*[101], it is challenging to determine whether this refers to the same animal since the Latin word *Delphinus* was used in the textual tradition for all Mediterranean regular species of family Delphinidae.

[94] Zoomathia research network: http://www.cepam.cnrs.fr/zoomathia/

[95] SourcEncyMe: http://atelier-vincent-de-beauvais.irht.cnrs.fr/encyclopedisme-medieval/programme-sourcencyme-corpus-et-sources-des-encyclopedies-medievales

[96] Ichtya: http://www.unicaen.fr/recherche/mrsh/document_numerique/projets/ichtya

[97] XML-TEI: http://www.tei-c.org/index.xml

[98] Biblissima Poject: http://www.biblissima-condorcet.fr/en

[99] Biblissima Onyology: http://doc.biblissima-condorcet.fr/ontologie-biblissima

[100] Hortus Sanitatis: https://www.unicaen.fr/puc/sources/depiscibus/accueil

[101] Naturalis Historia: https://en.wikipedia.org/wiki/Natural_History_(Pliny)

Furthermore, making sure that concepts share the same meaning across data sources requires the selection and/or definition of controlled and widely accepted vocabularies serving as semantic references with respect to taxonomical, cultural, geographical and chronological information.

TAXREF, the French zoological and botanical taxonomy [Gargominy et al., 2015] marks a large national and international scientific consensus. In the context of the Zoomathia network, it was selected to build a SKOS thesaurus supporting the integration of the considered heterogeneous data sources. Below, we first describe TAXREF, then we present the work achieved with xR2RML on the modelling and generation of a SKOS thesaurus based on TAXREF.

8.3.1 TAXREF: a Taxonomical Reference in Conservation Biology

As the national reference for nature and biodiversity, the French *National Museum of Natural History* (MNHN) is responsible for scientific and technical coordination of the natural heritage inventory. To this end, it maintains the *National Inventory of Natural Heritage*[102] (INPN), an information system that gathers contemporary occurrence data on fauna, flora and fungus of mainland France and overseas departments and collectivities. To support the integration of data from (currently) approximately 800 data sources, MNHN develops and distributes TAXREF [Gargominy et al., 2015], the French national taxonomical reference for fauna, flora and fungus. It is utilized by a large scope of public and private actors such as biologists, public collectivities, museum curators, architects, teachers, interested citizens, etc. TAXREF aims to:

(i) give an unambiguous unique scientific name for each taxon inventoried on the territory, that marks a national and international consensus (notably through the alignment with other international taxonomical references);
(ii) enable interoperability between databases in (archaeo)zoology and (archaeo)botany, to help the study of biodiversity and support strategies of natural heritage conservation; and
(iii) provide a follow-up on the taxonomic changes (synonymy, taxonomical hierarchy).

TAXREF is organized as a controlled, hierarchical list of scientific names. In its current 9th version, TAXREF inventories 485.189 taxa of living beings from the Paleolithic until now. The most salient fields provided for each taxon are as follows:

- *codeTaxon*: unique identifier of the scientific name;
- *codeReference*: identifier of the reference name. If *codeTaxon* is a reference name, then *codeTaxon* and *codeReference* are simply the same identifier. On the contrary, if codeTaxon identifies a synonym, then *codeReference* is the identifier of the reference name for that synonym;
- *codeParent*: identifier of the upper taxon in the classification;
- *libelleNom*: taxon scientific name;
- *libelleAuteur*: taxon authority (author name and publication year);
- *nomVernaculaire* and *nomVernaculaireAnglais*: French and English vernacular (common) names;
- *rang*: taxonomical rank represented by a code of two to four letters. Main ranks are classified as follows: domain, kingdom, phylum, division, class, order, family, tribe, genus, section, series, gender, species.

[102] Inventaire National du Patrimoine Naturel: http://inpn.mnhn.fr. Muséum National d'Histoire Naturelle [Ed]. 2003-2015.

- *habitat*: type of habitat in which the taxon usually lives (marine, freshwater, terrestrial...) coded as values from 1 to 8.
- A set of biogeographical statuses, one for each geographical region (mainland France and overseas departments and collectivities, “fr” stands for mainland France, “gua” for Guadeloupe, etc.). The status itself is a letter coded as follows: P stands for present, E for endemic, X for extinct, etc.

TAXREF can be browsed on the INPN Web site, downloaded in TSV format (tab-separated values), or queried using a Web service [Tercerie, 2016] that returns results in XML or JSON formats. As an example, Listing 16 shows a JSON excerpt describing the common dolphin whose scientific name is *Delphinus delphis*. Annotation `"habitat":1` states that it lives in a marine habitat, annotation `"rang":"ES"` states that the taxon belongs to the “species” taxonomical rank (“Espèce” in French). Annotations *"fr":"P"* and *"gua":"B"* represent its biogeographical status: they state that it is present in mainland France and occasional in Guadeloupe, a French overseas department. A comprehensive description of allowed values for the habitat, taxonomical rank and biogeographical status is provided in [Gargominy et al., 2015].

```
{
  "codeTaxon":"60878", "codeReference":"60878", "codeParent":"191591",
  "rang":"ES",
  "libelleNom":"Delphinus delphis",
  "libelleAuteur":"Linnaeus, 1758",
  "nomVernaculaire":"Dauphin commun à bec court, Dauphin commun",
  "nomVernaculaireAnglais":"Short-beaked common dolphin, Common Dolphin",
  "url":"http://inpn.mnhn.fr/espece/cd_nom/60878",
  "habitat":"1",
  "fr":"P", "gua":"B", "gf":"A", "mar":"B", "spm":"P",
  (...)
}
```

Listing 16: JSON description of the common dolphin, as returned by the TAXREF Web service

8.3.2 Modelling of a TAXREF-based SKOS Thesaurus

SKOS [Miles & Bechhofer, 2009], the *Simple Knowledge Organization System*, is a W3C standard designed to bridge the gap between existing controlled vocabularies, taxonomies and thesauri and the Semantic Web. Below, we present a work on the creation of a SKOS thesaurus faithfully representing the TAXREF taxonomical reference introduced above. The modelling of TAXREF in SKOS was achieved in collaboration with experts of TAXREF [Callou et al., 2015] and gave rise to conceptual discussions. Indeed, whereas TAXREF simply lists names, at the conceptual level, we must distinguish between taxa and names: a taxon represents a species independently of the names that can be used to refer to it. But as a convenience, we use the reference name to refer to the taxon. Consequently, taxa are the keystone of our modelling of TAXREF in SKOS: they are represented by SKOS concepts while the reference name and its synonyms are represented as SKOS labels.

Figure 15 depicts a simplified version of the SKOS modeling, exemplified with taxon *Delphinus delphis* and its synonym *Delphinus vulgaris*. Listing 17 provides a detailed representation of the target

modelling expressed in the RDF Turtle syntax. The complete RDF representation is split into two files available on the Morph-xR2RML GitHub repository[103]:

- The first file contains a definition of general annotations on the thesaurus (e.g. authors, license, version), domain properties (e.g. has habitat, has vernacular name, has biogeographical status) and domain concepts (e.g. taxonomical ranks, types of habitat), appropriately aligned with relevant properties and classes/concepts from other well-adopted ontologies and thesauri. These definitions were written manually.
- The second file contains the automatically generated representation of each taxon and associated names. This graph was materialized using Morph-xR2RML, as described in section 8.4.

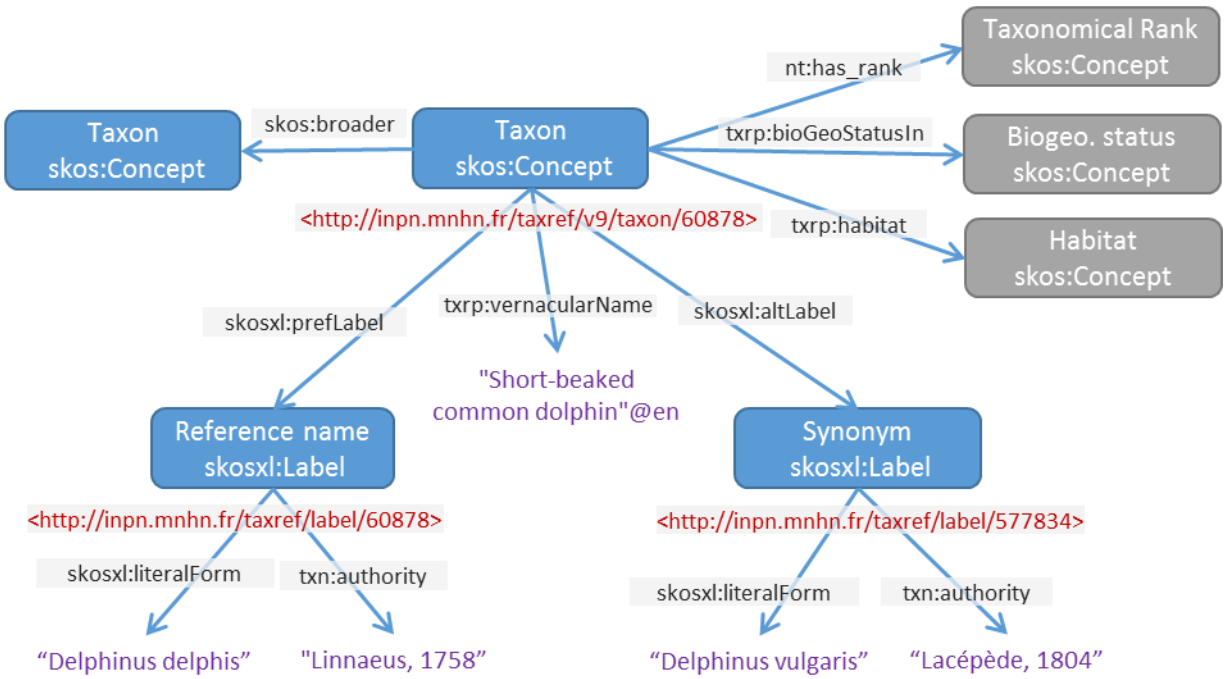


Figure 15: Model of a TAXREF-based SKOS Thesaurus.
A taxon is a SKOS concept. It is assigned a taxonomical rank, a biogeographical status and a habitat (top right). It has a parent taxon in the classification (skos:broader). The reference name and synonyms are SKOS-XL labels, they are assigned a literal form and an authority.

A taxon is represented by a SKOS concept (line 10) whose URI consists of the `http://inpn.mnhn.fr/taxref/9.0/taxon/` namespace followed by the taxon identifier (JSON field *codeTaxon*). The “9.0” value stands for the version of TAXREF: having the version in the URI is meant to track changes from one version to the other. The `skos:broader` property (line 13) models the relationships towards the parent taxon in the classification (*codeParent*). The reference name and its synonyms are modeled as SKOS labels (from the SKOS extension for Labels: SKOS-XL) referred to with properties `skosxl:prefLabel` and `skosxl:altLabel` respectively (lines 14-15). Their URIs consist of the `http://inpn.mnhn.fr/taxref/label` namespace followed by the taxon identifier (*codeTaxon*). Consequently, the taxon concept and the label representing its reference name shall use the same identifier in their URI, e.g. `http://inpn.mnhn.fr/taxref/9.0/taxon/60787` and `http://inpn.mnhn.fr/taxref/label/60787`. Conversely, the identifier of a synonym name is only used

[103] TAXREF SKOS: https://github.com/frmichel/morph-xr2rml/tree/master/morph-xr2rml-dist/example_taxref

in a label URI but never in a taxon URI. Unlike URIs of taxa, the TAXREF version is not part of label URIs: the reason is that scientific names (labels) do not change from one version to another, although they may be used alternatively as a reference or synonym name for a given taxon. The literal values of labels are defined with property `skosxl:literalForm` (lines 31 and 38). The taxonomical rank, habitat and biogeographical status are properties of the SKOS concept (lines 15-25), while the authorities and vernacular names are properties of SKOS labels (lines 28-30 and 35-37). Note that vernacular names are given as a comma-separated string in the JSON representation, whereas we wish to split this string in separate values with language tag “fr” or “en”.

Creating Linked Data. We identified existing vocabularies that can be reused in our context, keeping in mind that we wish to link the TAXREF thesaurus with existing, well-adopted data sets. We first focused on classes and properties that represent taxon characteristics (habitat, taxonomical rank, name authority). For example, taxonomical ranks are defined in various ontologies such as the NCBI taxonomic classification[104] and the GeoSpecies ontology[105]. Similarly, the type of habitat is commonly defined in several ontologies such as the ENVO environment ontology[106].

To keep full control over the TAXREF vocabulary, we defined SKOS concepts for the taxonomical ranks (e.g. lines 40-43), types of habitat (e.g. lines 45-48) and biogeographical statuses in namespace `http://inpn.mnhn.fr/taxref/`, and we aligned them with concepts from existing vocabularies using the `skos:exactMatch`, `skos:relatedMatch` or `skos:closeMatch` properties (lines 42-43, 47-48).

[104] NCBI Classification : http://www.ontobee.org/browser/index.php?o=NCBITaxon
[105] GeoSpecies: http://datahub.io/dataset/geospecies
[106] ENVO: http://www.ontobee.org/browser/index.php?o=ENVO

```
@prefix txrp:   <http://inpn.mnhn.fr/taxref/properties/> .
@prefix txrbgs: <http://inpn.mnhn.fr/taxref/bioGeoStatus#> .
@prefix nt:     <http://purl.obolibrary.org/obo/ncbitaxon#> .
@prefix dwc:    <http://rs.tdwg.org/dwc/terms/> .
@prefix txn:    <http://lod.taxonconcept.org/ontology/txn.owl#> .
@prefix dct:    <http://purl.org/dc/elements/1.1/> .
@prefix skos:   <http://www.w3.org/2004/02/skos/core#> .
@prefix skosxl: <http://www.w3.org/2008/05/skos-xl#> .

<http://inpn.mnhn.fr/taxref/9.0/taxon/60878> a skos:Concept ;
    skos:inScheme        <http://inpn.mnhn.fr/taxref/9.0/Taxref> ;
    skos:note            "Delphinus delphis" ;
    skos:broader         <http://inpn.mnhn.fr/taxref/9.0/taxon/191591> ;
    skosxl:prefLabel     <http://inpn.mnhn.fr/taxref/label/60878> ;
    skosxl:altLabel      <http://inpn.mnhn.fr/taxref/label/577834> ;
    txrp:habitat         <http://inpn.mnhn.fr/taxref/habitat#Marine> ;
    nt:has_rank          <http://inpn.mnhn.fr/taxref/taxrank#Species> ;
    txrp:bioGeoStatusIn  [ rdfs:label            "Metropolitan France" ;
                           dct:spatial           <http://sws.geonames.org/3017382/> ;
                           dwc:locationId        "TDWG:FRA; WOEID:23424819" ;
                           dwc:occurrenceStatus  txrbgs:P ] ;
    txrp:bioGeoStatusIn  [ rdfs:label            "Guadeloupe" ;
                           dct:spatial           <http://sws.geonames.org/3579143/> ;
                           dwc:occurrenceStatus  txrbgs:B ] .

<http://inpn.mnhn.fr/taxref/label/60878> a skosxl:Label ;
    txrp:isPrefLabelOf   <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> ;
    txn:authority        "Linnaeus, 1758" ;
    txrp:vernacularName  "Short-beaked common dolphin"@en, "Common Dolphin"@en,
                         "Dauphin commun"@fr, "Dauphin commun à bec court"@fr ;
    skosxl:literalForm   "Delphinus delphis" .

<http://inpn.mnhn.fr/taxref/label/577834> a skosxl:Label ;
    txrp:isAltLabelOf    <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> ;
    txn:authority        "Lacépède, 1804" ;
    txrp:vernacularName  "Common Dolphin"@en, "Short-beaked common dolphin"@en ,
                         "Dauphin commun à bec court"@fr, "Dauphin commun"@fr ;
    skosxl:literalForm   "Delphinus vulgaris" .

<http://inpn.mnhn.fr/taxref/taxrank#Species> a skos:Concept ;
    skos:prefLabel    "Species"@en ;
    skos:exactMatch   <http://purl.obolibrary.org/obo/NCBITaxon_species> ;
    skos:exactMatch   <http://rdf.geospecies.org/ont/geospecies#TaxonRank_species> .

<http://inpn.mnhn.fr/taxref/habitat#Marine> a skos:Concept ;
    skos:prefLabel    "Marine habitat"@en ;
    skos:relatedMatch <http://lod.taxonconcept.org/ontology/txn.owl#MarineHabitat> ;
    skos:exactMatch   <http://purl.obolibrary.org/obo/ENVO_00002227> .
```

Listing 17: SKOS representation of the "Delpinus delphis" taxon

8.4 Graph Materialization

To evaluate the capabilities of the xR2RML mapping language in a real-life use case, as well as the graph materialization mode of Morph-xR2RML with the connector to MongoDB, we operationalized the translation of the TAXREF taxonomical reference into an RDF graph with respect to the modelling presented in section 8.3.

The process is depicted in Figure 16. We first extracted the JSON representation of TAXREF v9.0 from the TAXREF Web service, and imported it into a MongoDB database as a collection *taxrefv9* of 485.189 documents (arrow "import"), each document accounting for a taxon. The rest of this section provides further details about the next steps, *i.e.* the design of the xR2RML mappings (section 8.4.1) and their execution (section 8.4.2) with Morph-xR2RML. Finally, we report an on-going experiment meant to expose TAXREF in SKOS as Linked Data (section 8.4.3).

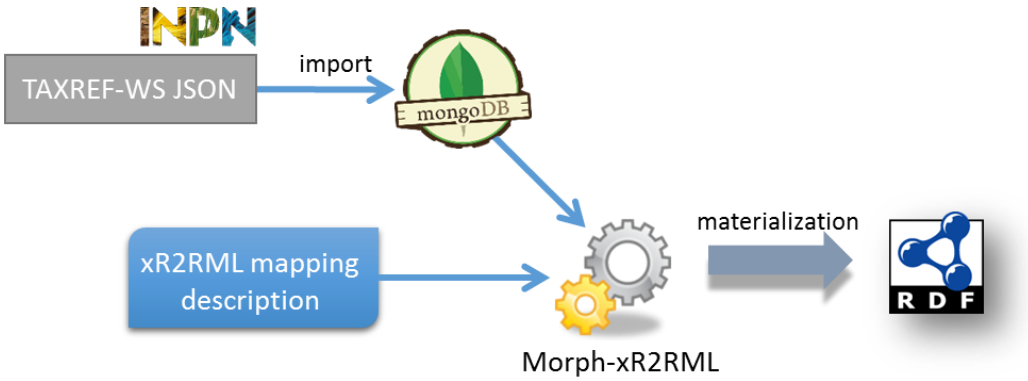


Figure 16: Translation of TAXREF in a SKOS Thesaurus

8.4.1 xR2RML Mapping for the TAXREF-based SKOS Thesaurus

The complete xR2RML mapping graph is provided in the xR2RML GitHub repository[107]. In this section, we briefly highlight noticeable questions encountered when writing the mappings.

As illustrated in Listing 16, the only way to figure out whether a scientific name is a reference name or a synonym is by comparing fields *codeTaxon* and *codeReference*. If these are the same, then *codeTaxon* is a reference name; otherwise, *codeTaxon* is a synonym of another reference name. This distinction shapes the logical source of the main triples maps of the xR2RML mapping graph. The logical source of triples maps about taxa and reference names (SKOS preferred label) use the MongoDB query:

```
db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )
```

whereas the logical source of triples maps about synonym names (SKOS alternate label) use the MongoDB query:

```
db.taxrefv9.find( { $where: 'this.codeTaxon != this.codeReference' } )
```

[107] xR2RML mapping graph for TAXREF v9 in SKOS: https://github.com/frmichel/morph-xr2rml/blob/master/morph-xr2rml-dist/example_taxref/xr2rml_taxref_v9.ttl

For instance, the triples map generating triples about the SKOS concept of each taxon is shown in Listing 18:

```
<#TM_Taxon>
  a rr:TriplesMap;
  xrr:logicalSource [ xrr:query
    """db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )""" ];
  rr:subjectMap [
    rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeTaxon}";
    rr:class skos:Concept
  ];
  rr:predicateObjectMap [
    rr:predicate skos:inScheme ;
    rr:objectMap [ rr:constant taxref:Taxref; rr:termType rr:IRI ]
  ];
  rr:predicateObjectMap [
    rr:predicate skos:broader;
    rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeParent}" ]
  ];
  rr:predicateObjectMap [
    rr:predicate skosxl:prefLabel;
    rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/label/{$.codeTaxon}" ]
  ].
```

Listing 18: xR2RML triples map generating triples about SKOS concept of each taxon

xR2RML Mixed Syntax Paths. Vernacular names are listed in the JSON fields `nomVernaculaire` for French, and `nomVernaculaireAnglais` for English. When multiple vernacular names exist, they are given as a comma-separated list. To split those values into individual RDF terms, we use the xR2RML mixed-syntax paths. For instance, the predicate-object map below extracts French vernacular names with JSONPath expression `$.nomVernaculaire` and selects the first and second names using the `CSV` constructor at indexes 0 and 1 respectively:

```
rr:predicateObjectMap [
  rr:predicate txrp:vernacularName;
  rr:objectMap [ xrr:reference "JSONPath($.nomVernaculaire)/CSV(0)"; rr:language "fr" ];
  rr:objectMap [ xrr:reference "JSONPath($.nomVernaculaire)/CSV(1)"; rr:language "fr" ]
];
```

String value `"Dauphin commun à bec court, Dauphin commun"` is thus translated into two RDF literals with the French language tag: `"Dauphin commun à bec court"@fr` and `"Dauphin commun"@fr`. If there is only one name in the field, then the first object map returns that name, and the second returns no value, thus generating no triple.

Custom Functions. The xR2RML mapping graph for TAXREF consists of 90 triples maps. This surprisingly high number is a consequence of the distance between the internal structure of TAXREF and the targeted SKOS modelling. We illustrate this distance with an example.

Habitats are coded in TAXREF with integer values, e.g. value ‘1’ represents the marine habitat, ‘2’ represents fresh water, etc. Translating the marine habitat into URI `<http://inpn.mnhn.fr/taxref/habitat#1>` would be straightforward using a template-valued term map that appends the value read from the database to the namespace

<http://inpn.mnhn.fr/taxref/habitat#>. Thus, a single triples map would be sufficient to generate all triples related to all types of habitat.

However, our modelling targets the generation of more meaningful URIs, such as <http://inpn.mnhn.fr/taxref/habitat#Marine>. This URI cannot be generated by a template; instead, we have to write a triples map whose query filters only taxa with habitat '1':

```
<#TM_Habitat_Marine>
  xrr:logicalSource [ xrr:query """
    db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference', 'habitat':'1' } )
  """ ];
  rr:subjectMap <#SM_Taxon>;
  rr:predicateObjectMap [
    rr:predicate txrfp:habitat;
    rr:objectMap [
      rr:constant <http://inpn.mnhn.fr/taxref/habitat#Marine>;
      rr:termType rr:IRI;
    ]
  ].
```

Consequently, we have to write one such triples map for each of the 8 habitat values. The same situation is observed for the 48 taxonomical ranks and 30 biogeographical statuses, that all result in dedicated triples maps.

Not only does this entail a cumbersome task of writing the manifold triples maps, but since each triples map translates the results of one MongoDB query, this also entails a much longer execution time: 90 queries are run, some of them returning tens or hundreds of thousands of documents. Ultimately, the same JSON documents are retrieved and parsed several times, each time for the generation of different triples.

This issue highlights the limitation aforementioned in section 5.8, regarding the management of custom functions within the xR2RML mapping language. In section 5.8 we suggested a possible extension of xR2RML leveraging CSVW and R2RML-F. Such an extension would make a notable difference in that particular case: instead of writing one triples map for each of the habitat values, a single triples map could use a *function-call-valued term map* (in the R2RML-F fashion) to reference a JavaScript function that would implement the mapping between habitat values and the corresponding strings that we want to use in the URIs.

8.4.2 Graph Materialization Processing

To perform the translation of TAXREF v9 into SKOS, we provisioned a virtual machine equipped with four logical cores and 8 GB RAM (the physical server has two 4-core 2.2GHz CPUs). Both the MongoDB database and the Java process running Morph-xR2RML ran on the same virtual machine. The Java virtual machine was allowed a maximum of 4 GB memory.

The translation of the 485.189 taxa ends up with an RDF graph of over 5.6 million triples. Along several runs, the translation process consistently required 35 to 38 minutes to complete.

Due to the distance between the SKOS modelling and the TAXREF JSON schema (discussed in the previous section), the execution of the 90 triples maps entails the execution of separate queries that may retrieve the same document several times to generate different types of triples. Indeed, the analysis of the execution log shows that approximately 2 million documents are retrieved. Hence, each document is processed roughly four times in average. This observation highlights even more the interest of considering CSVW and R2RML-F for the support of custom functions. The chart in Figure 17 shows the simultaneous CPU consumption of MongoDB and Morph-xR2RML during the graph materialization of TAXREF SKOS.

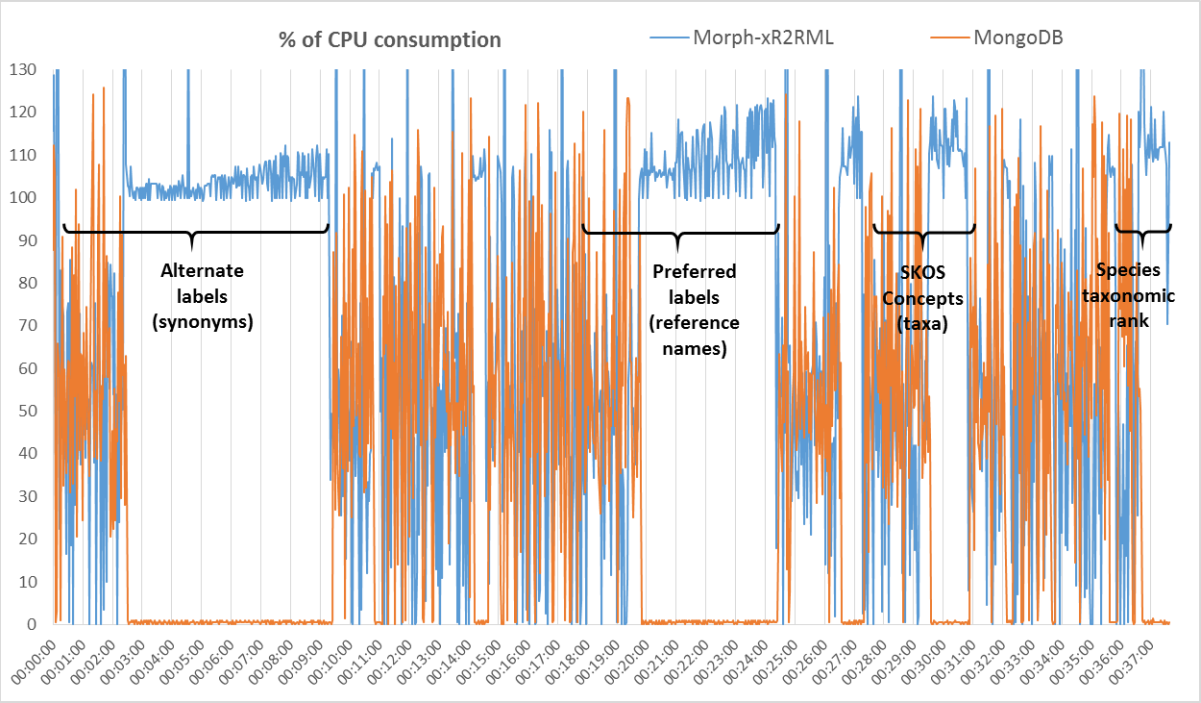


Figure 17: CPU (%) consumption during the graph materialization of TAXREF SKOS

A foregone observation is the correspondence between the intense CPU consumption of Morph-xR2RML (in blue) and the number of triples being materialized. Using the log traces to detect the processing start time and end time of each triples map, we have annotated the graph with some of the most salient phases. For instance, the phase labeled “Alternate labels (synonyms)” is the generation of the 257.965 SKOS alternate labels; it consists of two sub-phases. Firstly, MongoDB processes the query (00:00:18 to 00:02:16). Results are returned progressively to the client, which explains the activity of the blue line during that phase: it represents the activity of the API used to query MongoDB (Jongo), that retrieves and deserializes the resulting documents. Secondly, the API passes on the resulting documents to Morph-xR2RML that, in turn, materializes the triples (00:02:16 to 00:09:20). Unannotated phases correspond to triples maps that retrieve fewer documents. Given the time scale on the diagram, it is hardly possible to distinguish between the MongoDB querying phase and the generation of triples by Morph-xR2RML.

8.4.3 Perspectives

In the context of the Zoomathia research network, we aim to use the TAXREF-SKOS thesaurus to support multi-disciplinary studies on the history and transmission of zoological knowledge throughout

historical periods, combining the analysis of ancient and mediaeval literature, iconographic and archaeozoological resources. This will require the enrichment of the TAXREF-based thesaurus with philological and cultural information and its geographical extension to other Mediterranean areas (Greece, Italy, etc.). Furthermore, future works shall target the automatic discovery of links between taxa in TAXREF and equivalent scientific names in various other well-adopted open data sets and thesauri [Tounsi et al., 2015], may they be general knowledge data sources (*e.g.* DBpedia, national libraries) or domain specific data sources (*e.g.* Agrovoc, NCBI Organismal Classification, Vertebrate Taxonomical Ontology, Encyclopedia Of Life, etc.).

The representation of TAXREF in SKOS is not an end per se, but it should be envisaged as the enabler for future uses. In order for a large community to benefit from this work and to spur its adoption by developers of linked-data based applications, an on-going collaboration with the National Museum of Natural History seeks to set up a sustainable service capable of processing SPARQL queries and dereferencing TAXREF URIs. As a first step, the I3S laboratory has deployed an instance of the Corese-KGRAM Semantic Web factory [Corby & Zucker, 2010; Corby et al., 2012] that makes TAXREF-SKOS versions 8 and 9 accessible as two separate graphs. They may be accessed using a SPARQL 1.1 endpoint or as Linked Data, *i.e.* as dereferenced URI. Concretely, rewriting rules have been set up on the INPN Web server to implement content negotiation along with the appropriate redirections:

- If the client asks for HTML content, a URI such as <http://inpn.mnhn.fr/taxref/9.0/taxon/60878> is simply redirected to the existing Web page: https://inpn.mnhn.fr/espece/cd_nom/60878/
- If the client asks for an RDF syntax, the request is redirected to Corese-KGRAM. Under the hood, a SPARQL DESCRIBE query returns triples related to the URI.

Note: At the time of writing, this was successfully tested on a test server (inpn2.mnhn.fr), hence dereferenceable URIs have to start with inpn2.mnhn.fr for now. The solution shall be deployed live after a test phase.

SPARQL queries to https://inpn2.mnhn.fr/sparql are transparently redirected to the endpoint of Corese-KGRAM. Furthermore, Corese-KGRAM comes with a Web interface that allows to run SPARQL queries and display the result as HTML (depicted in Figure 18).

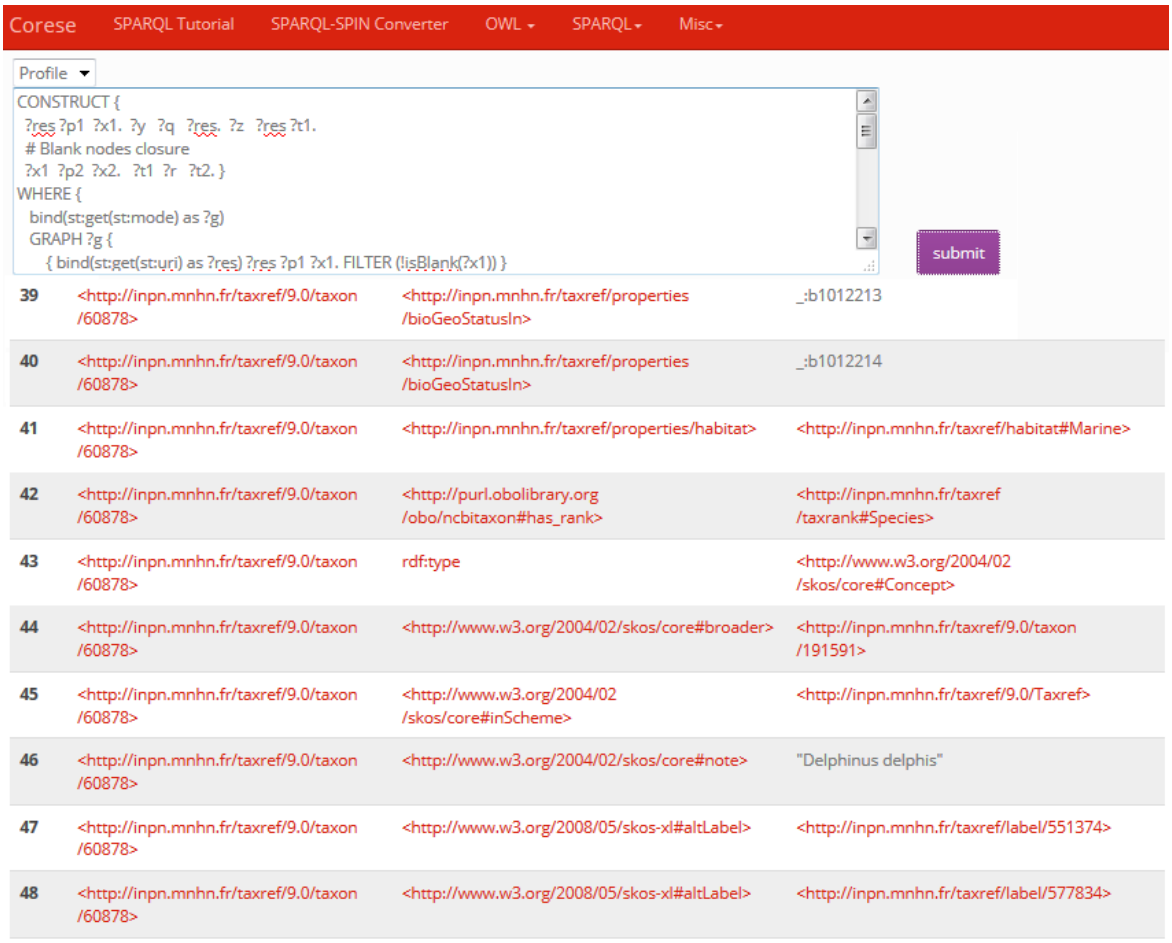


Figure 18: HTML rendering obtained when dereferencing the URI for taxon "Delphinus delphis"

8.5 SPARQL-to-MongoDB Query Translation

To evaluate the capabilities of the xR2RML-based SPARQL query rewriting method and the effectiveness of the abstract query optimizations, we used the MongoDB database mentioned in section 8.4. The database contains one JSON document for each taxon of the TAXREF v9 taxonomical reference.

In Chapter 7, we showed that Atomic Abstract Queries can be translated into equivalent MongoDB queries, but other operators of the abstract query language (INNER JOIN, LEFT OUTER JOIN, UNION) must be computed by the query-processing engine, *i.e.* Morph-xR2RML. Therefore, a first experimentation was conducted to assess the performances of Morph-xR2RML in the following situation: a SPARQL query consisting of one triple pattern bound to one triples map; the resulting Atomic Abstract Query is translated into a single MongoDB query (section 8.5.2). This gives us a way to estimate the time it takes to generate RDF triples independently of any other join or union operation.

Then, in a second experimentation, we measured the completion time of a set of SPARQL queries involving joins and/or unions, and we compared them to the time needed for a single triple pattern. Furthermore, we measured the gain obtained by performing optimizations at the level of the abstract query (section 8.5.3).

(a)
```
<#TM_Taxon>
  xrr:logicalSource [ xrr:query
   """db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )""";
   xrr:uniqueRef "$.codeTaxon" ];
  rr:subjectMap [
   rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeTaxon}";
   rr:class skos:Concept
  ];
  rr:predicateObjectMap [
   rr:predicate skos:broader;
   rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeParent}" ]
  ];
  rr:predicateObjectMap [
   rr:predicate skosxl:prefLabel;
   rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/label/{$.codeTaxon}" ]
  ].
```

(b)
```
<#SubjectMap_Taxon> a rr:SubjectMap;
  rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeTaxon}".
<#LogicalSource_Taxon> a xrr:LogicalSource;
  xrr:query
    """db.taxrefv9.find({$where: 'this.codeTaxon == this.codeReference'})""";
  xrr:uniqueRef "$.codeTaxon".

<#TM_Taxon_Type>
  xrr:logicalSource <#LogicalSource>; rr:subjectMap <# SubjectMap_Taxon>;
  rr:predicateObjectMap [
    rr:predicate rdf:type;
    rr:objectMap [ rr:constant skos:Concept; rr:termType rr:IRI ]
  ].
<#TM_Taxon_Broader>
  xrr:logicalSource <#LogicalSource>; rr:subjectMap <# SubjectMap_Taxon>;
  rr:predicateObjectMap [
    rr:predicate skos:broader;
    rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/9.0/taxon/{$.codeParent}"]
  ].
<#TM_Taxon_PrefLabel>
  xrr:logicalSource <#LogicalSource>; rr:subjectMap <# SubjectMap_Taxon>;
  rr:predicateObjectMap [
    rr:predicate skosxl:prefLabel;
    rr:objectMap [ rr:template "http://inpn.mnhn.fr/taxref/label/{$.codeTaxon}" ]
  ].
```

Listing 19: Normalization of one triples map (a) into three triples maps (b)

xR2RML mapping normalization. In a first step, we normalized the xR2RML mappings described in section 8.4.1. The normalization process (section 6.1.1) transforms the xR2RML mapping graph such that (i) a normalized triples map may contain only one predicate map and one object map, and (ii) an `rr:class` property in a subject map is transformed into an equivalent triples map with a constant predicate `rdf:type`. Listing 19 shows an example of a triples map (a) and the normalization thereof into three separate triples maps (b). The complete normalized TAXREF mapping is available on the repository of Morph-xR2RML[108].

8.5.1 Experimentation Environment

The query rewriting experimentations were conducted on a machine equipped with two 3.0GHz physical cores and 8 GB RAM. Both the MongoDB database and the Morph-xR2RML Java virtual machine were running on the same machine. The Java virtual machine was allowed a maximum of 4 GB memory. SPARQL queries were issued using two different tools: (i) queries returning few results were submitted with the Flint SPARQL editor[109]. When it comes to queries returning large result sets (in the order of tens of thousands of results), Flint becomes hardly usable because of the difficulty of Web browsers to print large formatted outputs. To work out this issue, we used a piece of JavaScript to submit a SPARQL query and redirect the output to a file.

8.5.2 Processing of a Simple Basic Graph Pattern

The experimentation reported in this section measures the performances of Morph-xR2RML in the case of a single triple pattern bound to one triples map, resulting in a single MongoDB query. We selected seven SPARQL SELECT queries (Q0 to Q6) tailored to produce an increasing number of results: from 1 result in Q0 to 227,224 results in Q6. In each case, one JSON document is translated into one RDF triple. Table 4 lists each query along with the corresponding triple pattern and semantics, the number of results it retrieves from the database, and the average time that Morph-xR2RML needed to process the query (the query processing spans the SPARQL query rewriting, the MongoDB query evaluation against the database, the RDF triples generation, as well as addition network overheads). For each query, 10 measures were performed: we report the average value and standard deviation. Besides, the last column gives the average processing time per result, that converges towards 0.48ms.

Figure 19 depicts the average query processing time as a function of the number of results (in blue). Morph-xR2RML builds on the Jongo API to process a MongoDB query. Therefore, we also performed a series of measures to estimate the time needed for Jongo to parse the query, pass it on to MongoDB and retrieve the results from MongoDB. Red dots represent the measures for Jongo/MongoDB only, while blue dots represent the measures for Morph-xR2RML. Dotted lines represent the linear regression lines of both series.

[108] https://github.com/frmichel/morph-xr2rml/blob/master/morph-xr2rml-dist/example_taxref_rewriting/xr2rml_taxrefv9-normalized.ttl

[109] Flint SPARQL Editor : http://openuplabs.tso.co.uk/demos/sparqleditor

Table 4: Execution time of SPARQL queries with one triple pattern

Query Id.	Query semantics and SPARQL triple pattern	No. results	Processing time ± std dev. (ms)	Avg. time per result (ms)
Q0	Find the reference name for taxon 60587 `?t skosxl:prefLabel <http://inpn.mnhn.fr/taxref/label/60587>`	1	451 ± 36	451.00
Q1	Get synonyms of taxon 95372 `<http://inpn.mnhn.fr/taxref/9.0/taxon/95372>` `skosxl:altLabel ?a`	164	522 ± 14	3.18
Q2	Get all bio-geographical statuses in St Pierre et Miquelon `?bgs dct:spatial <http://sws.geonames.org/3424932/>`	4 835	4,056 ± 65	0.84
Q3	Get all bio-geographical statuses in Guadeloupe `?bgs dct:spatial <http://sws.geonames.org/3579143/>`	17 956	9,665 ± 45	0.54
Q4	Get all bio-geographical statuses in New Caledonia `?bgs dct:spatial <http://sws.geonames.org/2139685/>`	35 703	17,289 ± 78	0.48
Q5	Get bio-geographical statuses in mainland France `?bgs dct:spatial <http://sws.geonames.org/3017382/>`	128 018	61,645 ± 671	0.48
Q6	Get all taxa (that are SKOS concepts) `?c a skos:Concept`	227 224	108,508 ± 459	0.48

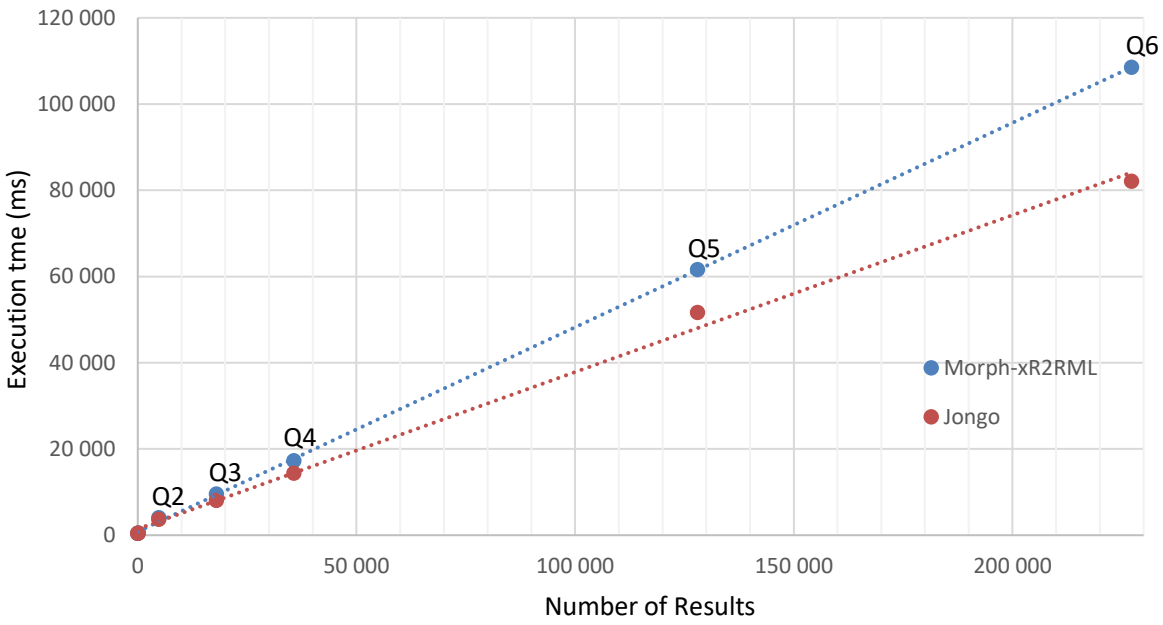


Figure 19: Average query processing time as a function of the number of results

From these two series, we estimated the overhead imposed by the query rewriting and triples generation of Morph-xR2RML, as compared to the sheer time required to run the query against the database. Figure 20 depicts this overhead for queries Q0 to Q6.

The confidence for Q0 and Q1, and to some extend for Q2, is very low as attested by the large error bars. Indeed, materializing a few triples is barely measurable (<1ms for Q0, and in the order of 30ms for Q1), such that the measure is very sensitive to environment variations.

Conversely, the confidence for Q3 to Q6 are quite high. Q3, Q4 and Q5 show a similar overhead of approximately 19%. Although we could expect the overhead percentage to be constant with larger numbers of results, it reaches 32% with Q6. A detailed analysis shows that the difference lies in the time needed to materialize the RDF triples. Compared to Q5, the number of results in Q6 increases by 77% while the materialization time increases by 120%. The variable term in Q3, Q4 and Q5 is a blank node whereas it is a URI in Q6. A tentative explanation is that the code of Morph-xR2RML is faster when producing blank nodes than when producing URIs (*e.g.* the production of URIs may entail additional index processing), unless this difference is in the Jena API on which Morph-xR2RML relies to handle RDF triples. Further works should consider using larger databases to assess this difference with more precision. In any case, the processing by Morph-xR2RML adds no more than 30% overhead to the time needed to query the database and retrieve the results.

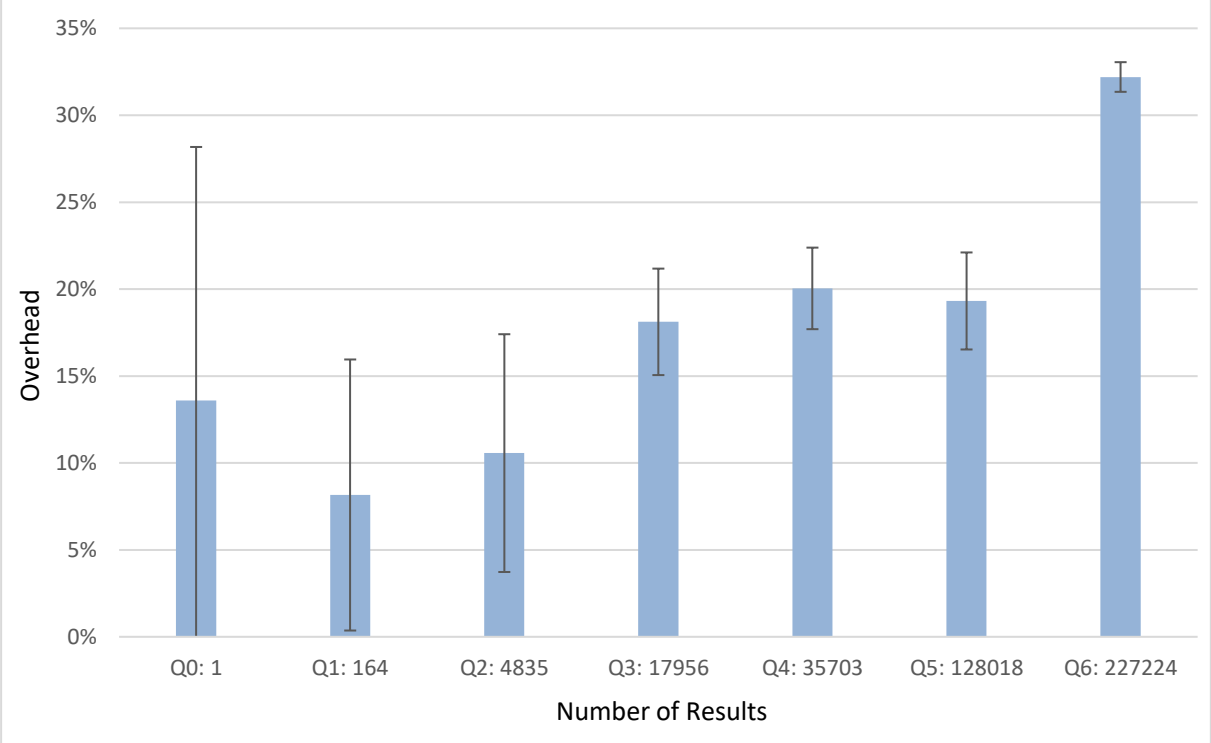


Figure 20: Overhead of Morph-xR2RML compared to a direct database query

Yet, waiting 10 seconds to get 18,000 results (query Q3) can be considered surprisingly long compared to native RDF triple stores. Firstly, let us notice that the MongoDB instance ran on the same machine as the Morph-xR2RML application, and that it was not tuned for the experimentation. It is likely that running MongoDB on a dedicated server and tuning appropriate indexes would improve the performances, but the gain might be mitigated by the additional network overhead.

Secondly, we compared we compared the time it takes to run a query (i) through the Jongo API (the case of Morph-xR2RML) and (ii) directly through MongoDB's own Java API. The results are presented in Figure 21 (the regression line is logarithmic). Surprisingly, it attests that, while Jongo is efficient for

few results (in the order of 100), it entails a significant overhead for larger results: 116% overhead for query Q6 (*i.e.* using Jongo more than doubles the query time). Jongo authors argue that the library is almost as fast as querying MongoDB directly, under the assumption that the marshalling/unmarshalling of JSON documents is left to Jongo. Morph-xR2RML retrieves JSON documents from Jongo as character strings in order to evaluate them with JSONPath expressions. Thus, it is possible that converting documents to strings hampers the performance of the library. Further investigation should be conducted to figure this out, keeping in mind that solving this issue could approximately save a factor 2 during the processing of large result sets.

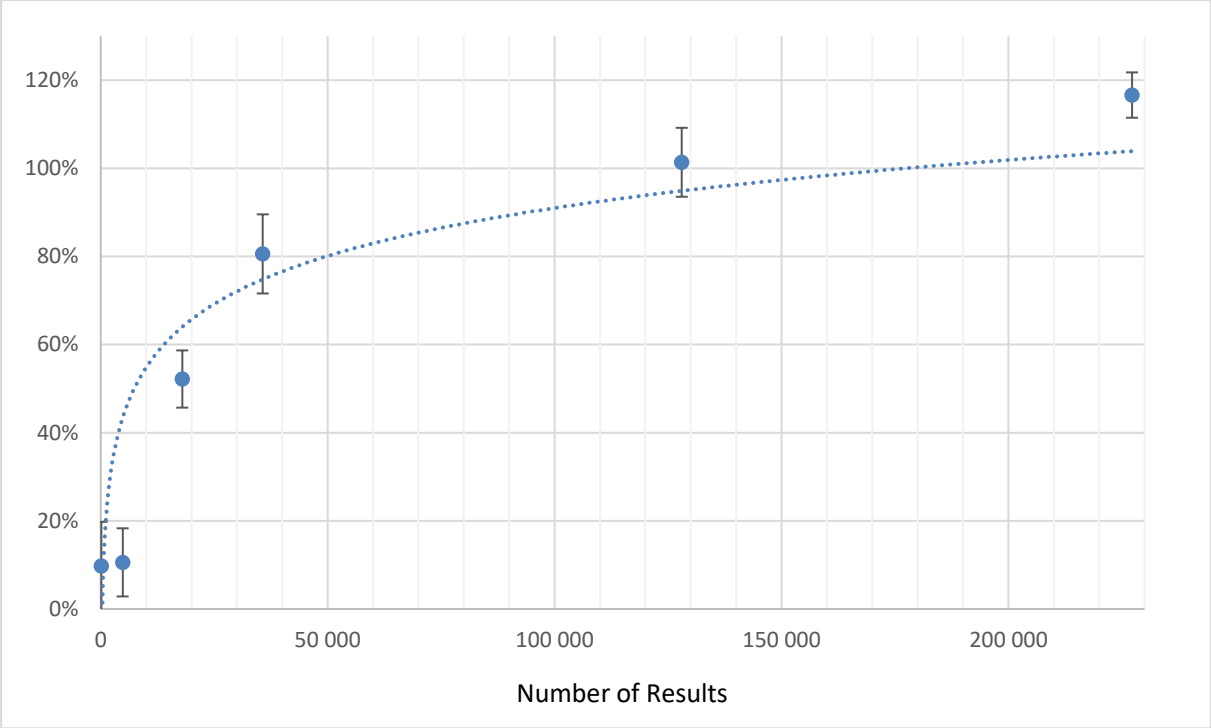


Figure 21: Overhead of querying MongoDB through the Jongo API

8.5.3 Joins, Unions and Query Optimizations

In our SPARQL-to-MongoDB rewriting method, each triple pattern of the SPARQL graph pattern is translated into an atomic query (or a union of atomic queries when multiple triples maps are bound to the triple pattern), and each atomic query is translated into a MongoDB query or a union of MongoDB queries. Operators of the abstract query language (INNER JOIN, LEFT OUTER JOIN, UNION, LIMIT, FILTER) are processed by the query processing engine, *i.e.* Morph-xR2RML.

Therefore, the completion time of a SPARQL query should be at least the sum of the completion time of each atomic query. In this matter, Figure 19 can help roughly assess the duration of an atomic query given the number of results it yields. In the rest of this section, we measure the completion time for a set of SPARQL queries involving joins and/or unions. Furthermore, we measure the gain obtained by performing optimizations at the level of the abstract query.

Notation. The example Atomic Abstract Queries depicted in this section are extracted from Morph-xR2RML execution logs. They present minor differences with the notation introduced in Chapter 6, as exemplified below. The first line gives the binding as a couple (triple pattern, bound triples map). When

several bindings share the same logical source (typically after a self-join elimination), several bindings may be listed in the same atomic query. The *from* part is the logical source of the triples map including the optional iterator, the *project* and *where* parts are respectively listed as a set of projections and a set of conditions, and the *Limit* part of an atomic query is denoted by "`LIMIT <integer>`".

```
{ Binding(?s ?p ?o -> TM_Taxon_Type)
  from   : db.taxrefv9.find( {$where:'this.codeTaxon==this.codeReference'} )
  project: Set($.codeTaxon AS ?s, rdf:type AS ?p, skos:Concept AS ?o)
  where  : Set(isNotNull($.codeTaxon))
} LIMIT 1000
```

8.5.3.1 Join, Self-Join

The SPARQL query below looks for taxa (variable ?t) that are present in the overseas collectivity of Saint-Pierre-et-Miquelon: `?t` must have a biogeographical status in this geographical location (`dct:spatial <http://sws.geonames.org/3424932/>`) with the status present (`dwc:occurrenceStatus taxrefbgs:P`). The graph pattern matches 12,708 triples that finally produce a result set of 4,236 solutions.

```
SELECT * WHERE {
    ?t taxrefprop:bioGeoStatusIn ?bgs .                    # triple pattern tp1
    ?bgs dct:spatial <http://sws.geonames.org/3424932/> . # triple pattern tp2
    ?bgs dwc:occurrenceStatus taxrefbgs:P .                # triple pattern tp3
}
```

Executed separately, the first triple pattern would be bound to 15 triples maps (one for each geographical location) and would yield 311,489 RDF triples; the second one would be bound to one triples map and would yield 4,835 triples, and the third one would be bound to 15 triples map and would yield 260,631 documents. The binding reduction step (section 6.5.3) removes all but one triples map bound to the first and third triple patterns. Under such reduced bindings, the query is translated into the join of three Atomic Abstract Queries depicted below. The first and second atomic queries yield 4,835 RDF triples while the third query yields 4,236 triples.

```
[ { Binding(tp1: ?t taxrefprop:bioGeoStatusIn ?bgs -> TM_SBG_SPM)
    from   : db.taxrefv9.find({$where:'this.codeTaxon == this.codeReference',
                               'spm':{$ne:''}, 'spm':{$ne:null}})
    project: Set($.codeTaxon AS ?t, $.codeTaxon AS ?bgs)
    where  : Set(isNotNull($.codeTaxon)) }
] INNER JOIN [
   [ { Binding(tp2: ?bgs dct:spatial http://sws.geonames.org/3424932/ -> TM_SBG_SPM_BN2)
       from   : db.taxrefv9.find({$where:'this.codeTaxon == this.codeReference',
                                  'spm':{$ne:''}, 'spm':{$ne:null}})
       project: Set($.codeTaxon AS ?bgs)
       where  : Set(isNotNull($.codeTaxon)) }
   ] INNER JOIN [
     { Binding(tp3: ?bgs dwc:occurrenceStatus taxrefbgs:P ->TM_SBG_SPM_BN1)
       from   : db.taxrefv9.find({$where:'this.codeTaxon == this.codeReference',
                                  'spm':{$ne:''}, 'spm':{$ne:null}})
       project: Set($.codeTaxon AS ?bgs)
       where  : Set(isNotNull($.codeTaxon), equals($.spm, P)) }
   ] ON Set(?bgs)
] ON Set(?bgs)
```

Executed with no binding reduction, the query completes in almost 10 minutes (600s). With the binding reduction, the query completes in 8.53s in average, the querying to MongoDB accounts for 47% of this total time, the generation of the RDF triples accounts for 11% and the processing of joins for 39%.

A closer look to that query shows that it contains two self-joins that can be eliminated (see section 6.7.3): (i) all three queries share the same *From* part (the logical source), (ii) they are joined on the `?bgs` variable that is always projected from the same reference `$.codeTaxon`, and (iii) `$.codeTaxon` is declared as a unique identifier in at least one triples map bound to the three triple patterns (with property `xrr:uniqueRef`). This self-join elimination entails an optimized query that now consists of a single atomic query, as shown below. Note that the *project* and *where* parts have been merged, and the three bindings now apply to this atomic query: the same target query is used to generate RDF triples matching the three triple patterns.

```
{ Binding(tp1: ?t   taxrefprop:bioGeoStatusIn ?bgs -> TM_SBG_SPM),
  Binding(tp2: ?bgs dct:spatial http://sws.geonames.org/3424932/ -> TM_SBG_SPM_BN2),
  Binding(tp3: ?bgs dwc:occurrenceStatus taxrefbgs:P -> TM_SBG_SPM_BN1)
  from   : db.taxrefv9.find({$where:'this.codeTaxon == this.codeReference',
                             'spm':{$ne:''}, 'spm':{$ne:null}})
  project: Set($.codeTaxon AS ?t, $.codeTaxon AS ?bgs)
  where  : Set(isNotNull($.codeTaxon), equals($.spm, P))
}
```

This optimized query completes in 2,966ms in average, *i.e.* a 65% gain compared to the non-optimized query.

8.5.3.2 Union, Self-Union

The SPARQL query below seeks biogeographical statuses that are either absent (A) or disappeared (D), in any of the territories covered by TAXREF. It produces one solution for each of the 13,975 triples matched by the graph pattern.

```
SELECT * WHERE {
    { ?bgs dwc:occurrenceStatus taxrefbgs:A . }    # triple pattern tp1
    UNION
    { ?bgs dwc:occurrenceStatus taxrefbgs:D . }    # triple pattern tp2
}
```

There exist 15 different geographical locations spanning mainland France and overseas collectivities, represented by 15 triples maps in our mapping. Consequently, each triple pattern is translated into a union of 15 per-triples-map atomic queries. Overall, the graph pattern is translated into a union of 30-per-triples map atomic queries. This is exemplified with two regions in the abstract query below: the first and second atomic queries are related to the first triple pattern (status absent), the third and fourth queries to the second triple pattern (status disappeared).

```
[{ Binding(tp1: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                     http://inpn.mnhn.fr/taxref/bioGeoStatus#A -> TM_SBG_Guadeloupe_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'gua':{$ne: ''}, 'gua': {$ne: null} } )
  project: Set($.codeTaxon AS ?bgs)
  where   : Set(NotNull($.codeTaxon), Equals($.gua, A)) }
] UNION [
{ Binding(tp1: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#A -> TM_SBG_SPM_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'spm': {$ne: ''}, 'spm': {$ne: null} } )
  project: Set($.codeTaxon AS ?bgs)
  where   : Set(NotNull($.codeTaxon), Equals($.spm, A)) }
] UNION [
{ Binding(tp2: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#D -> TM_SBG_Guadeloupe_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'gua': {$ne: ''}, 'gua': {$ne: null} } )
  project: Set($.codeTaxon AS ?bgs)
  where   : Set(NotNull($.codeTaxon), Equals($.gua, D)) }
] UNION [
{ Binding(tp2: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#D -> TM_SBG_SPM_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'spm': {$ne: ''}, 'spm': {$ne: null} } )
  project: Set($.codeTaxon AS ?bgs)
  where   : Set(NotNull($.codeTaxon), Equals($.spm, D)) }
] UNION (...)
```

In this query, each triples map related to a geographical location (`TM_SBG_Guadeloupe_BN1`, `TM_SBG_SPM_BN1`) is used twice, once for each status (A or D). The self-union elimination detects this and merges the queries that have the same logical source (*i.e.* the same geographical location in this case). The optimized query consists of one atomic query for each region, each query being relevant for two bindings. Note that the conditions of each atomic query is the logical OR of the conditions of the merged queries. This is illustrated with the two regions in the abstract query below.

```
[{Binding(tp1: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#A -> TM_SBG_Guadeloupe_BN1),
  Binding(tp2: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#D -> TM_SBG_Guadeloupe_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'gua': {$ne: ''}, 'gua': {$ne: null} } )
  project: Set($.codeTaxon AS ?bgs)
  where   : Set(Or(And(NotNull($.codeTaxon), Equals($.gua, A)),
                   And(NotNull($.codeTaxon), Equals($.gua, D)))) }
] UNION [
{ Binding(tp1: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#A -> TM_SBG_SPM_BN1),
  Binding(tp2: ?bgs @http://rs.tdwg.org/dwc/terms/occurrenceStatus
                    http://inpn.mnhn.fr/taxref/bioGeoStatus#D -> TM_SBG_SPM_BN1)
  from    : db.taxrefv9.find( {$where: 'this.codeTaxon == this.codeReference',
                               'spm': {$ne: ''}, 'spm': {$ne: null} } )
```

```
  project: Set($.codeTaxon AS ?bgs)
  where  : Set(Or(And(NotNull($.codeTaxon), Equals($.spm, A)),
                  And(NotNull($.codeTaxon), Equals($.spm, D)))) }
] UNION (...)
```

This query completes in 39.5s in average, while the non-optimized query completes in 40.9s in average. This difference can seem surprisingly small. It is in fact the result of another, more technical optimization: during the processing of an abstract query, Morph-xR2RML stores the results of intermediate MongoDB queries into a memory cache in order to speed up further atomic queries that may have the same logical source. In the optimized query, each MongoDB query is executed once and its results are used immediately to materialize two triples maps. In the non-optimized query on the other hand, the two MongoDB queries are also executed just once but their results are used at two different times: the small overhead is due to the time it takes to retrieve the query results from the memory cache.

More generally, unlike the self-join elimination that is likely to save the generation of possibly many triples and the computation of a large join, the self-union elimination is likely to have a more limited impact on performances thanks to this technical optimization. Besides, whether a self-union is eliminated or not, the query shall always generate the same number of resulting triples.

8.5.3.3 Constant Projection

The two SPARQL queries below are prototypical of schema exploration. For instance, the Flint SPARQL editor runs both of them when the user first selects an endpoint.

```
SELECT DISTINCT ?p WHERE {?s ?p ?o} ORDER BY ?p LIMIT 1000
SELECT DISTINCT ?o WHERE {?s a ?o} ORDER BY ?o LIMIT 1000
```

Naively, the first query is translated into an abstract query that makes the UNION of all per-triples-map atomic queries. In our context, this amounts to a UNION of 150 atomic queries, as illustrated below. Even though some self-joins could be eliminated, executing this query would materialize the whole database.

```
[
{ Binding(?s ?p ?o -> TM_Taxon_Type)
  from   : db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )
  project: Set($.codeTaxon AS ?s, http://www.w3.org/1999/02/22-rdf-syntax-ns#type AS ?p,
               http://www.w3.org/2004/02/skos/core#Concept AS ?o)
  where  : Set(isNotNull($.codeTaxon)) }
LIMIT 1000
] UNION [
{ Binding(?s ?p ?o -> TM_Taxon_PrefLabel)
  from   : db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )
  project: Set($.codeTaxon AS ?s, http://www.w3.org/2008/05/skos-xl#prefLabel AS ?p,
               $.codeTaxon AS ?o)
  where  : Set(isNotNull($.codeTaxon)) }
LIMIT 1000
] UNION
{ Binding(?s ?p ?o -> TM_TaxonomicalRank_Species)
  from   : db.taxrefv9.find({$where: 'this.codeTaxon==this.codeReference', 'rang':'ES'} )
  project: Set($.codeTaxon AS ?s, http://purl.obolibrary.org/obo/ncbitaxon#has_rank AS ?p,
```

```
              http://inpn.mnhn.fr/taxref/taxrank#Species AS ?o)
  where  : Set(isNotNull($.codeTaxon)) }
LIMIT 1000
] UNION [
(...)
UNION [
{ Binding(?s ?p ?o -> TM_Habitat_Marine)
  from   : db.taxrefv9.find( {$where:'this.codeTaxon==this.codeReference', 'habitat':'1'} )
  project: Set($.codeTaxon AS ?s, http://inpn.mnhn.fr/taxref/properties/habitat AS ?p,
              http://inpn.mnhn.fr/taxref/habitat#Marine AS ?o)
  where  : Set(isNotNull($.codeTaxon)) }
LIMIT 1000
]
```

However, it occurs that the only projected variable, ?p, is always bound to constant term maps: variable ?p is never built using values from the database, but always comes from constant predicate maps *e.g.* "`rr:predicate skosxl:prefLabel`". Hence, there is no need to query the database to get the values of ?p.

The *Constant Projection* optimization rewrites the initial query into the query below that can be evaluated without even querying the database:

```
SELECT DISTINCT ?p WHERE { VALUES ?p (
        <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
        <http://www.w3.org/2008/05/skos-xl#prefLabel>
        <http://purl.obolibrary.org/obo/ncbitaxon#has_rank>
        <http://inpn.mnhn.fr/taxref/properties/habitat>
        <http://www.w3.org/2000/01/rdf-schema#label>
        <http://www.w3.org/2008/05/skos-xl#literalForm>
        <http://lod.taxonconcept.org/ontology/txn.owl#authority>
        <http://inpn.mnhn.fr/taxref/properties/isAltLabelOf>
        <http://rs.tdwg.org/dwc/terms/locationId>
        <http://www.w3.org/2008/05/skos-xl#altLabel>
        <http://inpn.mnhn.fr/taxref/properties/bioGeoStatusIn>
        <http://inpn.mnhn.fr/taxref/properties/vernacularName>
        <http://inpn.mnhn.fr/taxref/properties/isPrefLabelOf>
        <http://rs.tdwg.org/dwc/terms/occurrenceStatus>
        <http://www.w3.org/2004/02/skos/core#broader>
        <http://lod.taxonconcept.org/ontology/txn.owl#habitat>
        <http://purl.org/dc/elements/1.1/spatial>
        <http://www.w3.org/2004/02/skos/core#note>
) } ORDER BY ?p LIMIT 1000
```

8.5.3.4 Filter Propagation

The SPARQL query below retrieves the taxon (variable ?t) with preferred label `<http://inpn.mnhn.fr/taxref/label/60585>` alongside two different alternate labels (variables ?a and ?b).

```
SELECT * WHERE {
    ?t skosxl:prefLabel <http://inpn.mnhn.fr/taxref/label/60585> .
    ?t skosxl:altLabel ?a .
    ?t skosxl:altLabel ?b .
    FILTER (?a != ?b)
```

```
        }
```

In a first step, it is translated into the inner join of three atomic abstract queries. The first atomic query retrieves 1 document while the second and third queries retrieve 257,965 documents.

```
[{ Binding(?t skosxl:prefLabel http://inpn.mnhn.fr/taxref/label/60585->TM_Taxon_PrefLabel)
  from   : db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )
  project: Set($.codeTaxon AS ?t)
  where  : Set(isNotNull($.codeTaxon), equals($.codeTaxon, 60585)) }
] INNER JOIN [
    [{ Binding(?t skosxl:altLabel ?a -> TM_Taxon_AltLabel)
      from   : db.taxrefv9.find( { $where: 'this.codeTaxon != this.codeReference' } )
      project: Set($.codeReference AS ?t, $.codeTaxon AS ?a)
      where  : Set(isNotNull($.codeReference), isNotNull($.codeTaxon)) }
    ] INNER JOIN [
    { Binding(?t skosxl:altLabel ?b -> TM_Taxon_AltLabel)
      from   : db.taxrefv9.find( { $where: 'this.codeTaxon != this.codeReference' } )
      project: Set($.codeReference AS ?t, $.codeTaxon AS ?b)
      where  : Set(isNotNull($.codeReference), isNotNull($.codeTaxon)) }
    ] ON Set(?t)
] ON Set(?t)
```

Executed naively, the inner-most join computes the join of 257,965 documents with another 257,965 triples generated from the same database documents. With a smarter ordering of the joins, the triple produced by the first atomic query is joined with the 257,965 triples of the second one to produce two triples (taxon 60585 has two synonyms), that, in turn, are joined with the 257,965 triples of the third query. Yet, two joins of 257,965 triples with one then two triples have to be performed. Some tests show that the time needed to complete this query is in the order of 4 minutes.

The *Filter Propagation* optimization leverages some situations where, within a join of two queries, a condition on a shared variable can be propagated from one query to the other. In the example, the two joins are performed on variable `?t`. The first atomic query projects `?t` as `$.codeTaxon` and has condition `equals($.codeTaxon, 60585)`. In the second and third queries, variable `?t` is projected as `$.codeReference`. Therefore, the join condition can only be satisfied if reference `$.codeReference` returns the value 60585. In other words, we propagate the condition `equals($.codeTaxon, 60585)` into the second and third queries as `equals($.codeReference, 60585)`. The optimized abstract query is now as follows:

```
[{ Binding(?t skosxl:prefLabel http://inpn.mnhn.fr/taxref/label/60585->TM_Taxon_PrefLabel)
  from   : db.taxrefv9.find( { $where: 'this.codeTaxon == this.codeReference' } )
  project: Set($.codeTaxon AS ?t)
  where  : Set(isNotNull($.codeTaxon), equals($.codeTaxon, 60585)) }
] INNER JOIN [
    [{ Binding(?t skosxl:altLabel?a -> TM_Taxon_AltLabel)
      from   : db.taxrefv9.find( { $where: 'this.codeTaxon != this.codeReference' } )
      project: Set($.codeReference AS ?t, $.codeTaxon AS ?a)
      where  : Set(isNotNull($.codeTaxon), equals($.codeReference, 60585)) }
    ] INNER JOIN [
    { Binding(?t skosxl:altLabel ?b -> TM_Taxon_AltLabel)
      from   : db.taxrefv9.find( { $where: 'this.codeTaxon != this.codeReference' } )
      project: Set($.codeReference AS ?t, $.codeTaxon AS ?b)
      where  : Set(isNotNull($.codeTaxon), equals($.codeReference, 60585)) }
```

```
    ] ON Set(?t)
] ON Set(?t)
```

Consequently, the second and third queries only produce two RDF triples. Once translated to MongoDB, the execution of this query lasts 565ms in average, that is a gain factor in the order or 400.

8.6 Conclusions and Perspectives

In this chapter, we showed that the approach defined from Chapter 5 to Chapter 7 is effective, as it allows to translate a NoSQL database into RDF and query it using SPARQL, relying on xR2RML to describe the mapping of database documents to an RDF representation. We illustrated this with the example of the MongoDB database, and the concrete use case of a taxonomical reference that we translated into a SKOS thesaurus. We showed that the Morph-xR2RML prototype implementation effectively achieves both the graph materialization and SPARQL query rewriting approaches. We measured the performances of several example SPARQL queries, but the lack of benchmark or other SPARQL-to-MongoDB implementations hinders further comparison.

Our experimentations underlined some limitations of the Morph-xR2RML prototype that can impair performances. Below we discuss them and suggest ways that could help improve performances.

Direct MongoDB query vs. Jongo API. In section 8.5.2, we showed that using the Jongo API to query the database presents an overhead that becomes dramatic as the number of results increases (over 100% beyond 200.000 results). Besides, different tests showed that querying MongoDB often takes in the order of 50% of the overall query processing time. Thus, solving this issue could save in the order of 25% of the total query processing time.

Lazy Generation of RDF Triples*.* Morph-xR2RML generates the RDF triples resulting from each of the atomic queries, and performs joins afterwards (INNER JOIN, LEFT OUTER JOIN). In some cases, the join query may rule out many of the triples that were just materialized. A possible improvement would consist in evaluating joins on the database documents, thereby generating RDF triples only at a later stage, when unnecessary documents have already been ruled out.

MongoDB find vs. aggregate Queries. In section 3.2, we mentioned that Botoeva et al. have proposed a generalization of the OBDA principles to support MongoDB [Botoeva et al., 2016a], in which they produce MongoDB *aggregate* queries. This guarantees a semantics-preserving SPARQL 1.0 to MongoDB query translation, thus making the query translation much easier. In practice however, *aggregate* pipelines may perform poorly. To optimize them, an option suggested by the authors is to decompose the pipeline into smaller queries and have the query-processing engine perform the remaining steps. Our approach works the other way around: it produces less-expressive MongoDB *find* queries, leaving much more work to the query-processing engine, but making it easier to implement smart optimizations.

Some preliminary tests indicate that *aggregate* queries indeed perform faster that *find* queries in various cases. Consequently, it would be interesting to characterize how to split an abstract query into *aggregate* or *find* queries. However, unlike ontop, xR2RML allows for rich JSONPath expressions to evaluate a JSON document and generate RDF terms. Typically, a SPARQL variable may be bound not

only to a simple JSON field, but to rich expressions such as the "`$.members[?(@.age>=40)].name AS ?senior`" example in the running example in Chapter 7. In this matter, further studies should rely on the translation algorithm in section 7.4 to figure out how to translate such JSONPath expressions into *aggregate* queries.

Chapter 9. Conclusions and Perspectives

9.1 Summary

As more institutions and organizations around the world publish increasing amounts of Linked Data, a new layer of the Web surfaces: the Web of Data. Meanwhile, data integration techniques increasingly rely on RDF as a pivot format. Indeed, RDF allows building on the manifold existing domain knowledge formalizations, it makes it possible to use the Semantic Web reasoning capabilities, and it helps leverage the huge knowledge base represented by the Web of Data.

To a great extent, RDF-based data integration, and in particular the success of Web of Data, depend on the ability to reach out the legacy data that are locked in closed silos. In the last 15 years, various works have tackled the challenge of translating structured data into RDF, starting with relational databases, and the CSV/TSV and XML data formats. Meanwhile, the overwhelming success of NoSQL databases has made the database landscape more heterogeneous than ever. So far, though, these data remain inaccessible to RDF-based data integration systems, and although some of them may be of interest to a large audience, they remain invisible to the Web of Data.

Therefore, to harness the potential of NoSQL databases and more generally non-RDF data sources, the objective of this thesis is to enable RDF-based data integration over heterogeneous databases and, in particular, to bridge the gap between the Semantic Web and the NoSQL family of databases.

In **Chapter 3,** we first reminded general data integration principles and specifically ontology-based data integration principles. Since our goal is to enable RDF-based data integration over heterogeneous data sources, we analyzed previous works on the translation of existing data formats and databases into RDF, and we made a focus on the keystone of all these approaches: the description of mappings between a data source and a global schema (typically in the form of domain ontologies).

In **Chapter 4**, we carried on this analysis towards the definition of the requirements that a generalized mapping method should meet, in order to map data sources to an arbitrary RDF representation. These requirements are:

- Openness: accept any query language to query the target database, and accept any syntax to refer to data items (such as column names, XPath, JSONPath),
- Generate RDF collections (lists) and containers (bags, sequences, alternates),
- Model join queries to retrieve cross-referenced logical entities,
- Implement domain logic by means of custom functions logic,
- Produce RDF triples in named graphs.

Then, we evaluated existing mapping languages with respect to these requirements, and we pinpointed some of them as valuable groundwork to underpin a generalized mapping language.

Armed with this set of requirements, in **Chapter 5** we presented our first contribution *i.e.* the specification of xR2RML, a generalized mapping language to map data sources with heterogeneous data models and query capabilities to RDF, in a transparent and flexible manner. xR2RML builds upon

and extends R2RML, the W3C standard for relational databases, and RML to address the mapping of heterogeneous data formats.

To foster the development of SPARQL interfaces to heterogeneous databases, our second contribution is defined in **Chapter 6**. This is the first step of a two-step approach to execute SPARQL queries over heterogeneous databases, utilizing xR2RML to describe the mapping of a target database to RDF. We defined a pivot, abstract query language, meant to make the approach independent of the specificities of the target database. The language derives from the syntax and semantics of SPARQL, and embeds the new concept of *Atomic* A*bstract Query* which is the result of matching a SPARQL triple pattern with appropriate xR2RML mappings. Then, leveraging R2RML-based SPARQL-to-SQL works, we proposed a method to translate a SPARQL graph pattern into an abstract query. The method determines a reduced set of relevant mappings for each SPARQL triple pattern, and several optimizations are enforced in order to produce an efficient abstract query.

The second step of our two-step approach enacts the database-dependent phase. It translates an abstract query into a target database query. To demonstrate the effectiveness of this overall approach, and to illustrate the effort it takes to translate from SPARQL towards a query language with a lower expressiveness, in **Chapter 7** we selected the popular NoSQL database MongoDB as an example, and we devised a translation method from abstract queries into MongoDB queries. This is our third contribution. We showed that the discrepancy between the expressiveness of SPARQL queries and MongoDB *find* queries makes it impossible to ensure that query semantics be preserved. Nevertheless, we proved that our method retrieves all matching documents, in addition to possibly non-matching ones that we rule out in a last step to guarantee that the result is correct.

Lastly, in **Chapter 8**, we first shortly described the prototype implementation of our method. Then, we reported the results of an experimental evaluation carried out in the context of a real-world use case, where we translated the TAXREF taxonomical reference stored in a MongoDB database into a SKOS thesaurus. The evaluation illustrates the utilization of some advanced features of xR2RML, and presents performance measures with respect to the RDF graph materialization and the SPARQL-to-MongoDB query rewriting.

From a broader perspective, we have shown that translating a SPARQL query into efficient concrete queries can be challenging when it comes to address data sources such as NoSQL databases. These systems are generally optimized for fast storage and retrieval of vast collections of documents. They favor scalability, high throughput and availability over consistency and query language expressiveness. Consequently, they often come with denormalized data models where redundancy is common, and barely support joins. Hence, it is likely that the hurdles we encountered with MongoDB will be encountered with other NoSQL databases alike.

9.2 Perspectives

Along this manuscript, we have highlighted various challenges that still need to be addressed. Below, we remind and discuss them in a more global perspective.

Producing Efficient Queries. We have shown that translating a SPARQL query into efficient concrete queries is difficult in the general case. In the case of MongoDB, the processing of joins is shifted to the query processing engine, and can result in poor performances when joined subqueries are not selective enough. Furthermore, real-world SPARQL queries often contain large graph patterns with many joined triple patterns. It is therefore critical to be able to process joins efficiently. Thus, beyond the optimizations that we implemented at the abstract query level, query-plan optimization techniques shall be investigated to help answer the following questions:

- Can we rewrite a SPARQL graph pattern in a way that facilitates the production of an efficient abstract query?
- How to inject intermediate results into a subsequent query, similar to the bind join optimization [Haas et al., 1997]?
- How to reorder joins based on the number of results of subqueries, similar to methods proposed in distributed query engines? [Schwarte et al., 2011; Görlitz & Staab, 2011; Macina et al., 2016]
- Can we perform lazy evaluation of joins by progressively materializing triples on each side of the join until the expected number of results is reached? This would typically resemble the method employed in the non-blocking evaluation of queries in the context of Triple Pattern Fragments [Verborgh et al., 2016].

Regarding the specific case of MongoDB, other leads should be studied. We wish to explore the rewriting from our abstract query language to MongoDB *aggregate* queries, as advocated in [Botoeva et al., 2016a], and characterize mappings with respect to the type of query that shall perform best: single vs. multiple separate queries, *find* vs. *aggregate*, and figure out a balance between the two approaches.

Improving the Abstract-to-MongoDB Translation. Below we discuss several improvement leads that could be investigated to overcome the limitations of the translation from the abstract query language to MongoDB. In section 8.6, we discussed several leads with respect to the use of direct MongoDB query vs. Jongo API, the lazy generation of RDF Triples and the use. Of MongoDB find vs. aggregate queries. Additionally, our implementation of xR2RML for MongoDB relies on JSONPath to extract data elements from MongoDB results. In turn, the SPARQL rewriting process must handle conditions on JSONPath expressions. It must cope with the expressiveness discrepancy between SPARQL and MongoDB, and between JSONPath and MongoDB alike. While the earlier is necessary (our goal is specifically to access heterogeneous databases with SPARQL), the latter is somewhat more an implementation choice. Hence, an investigation should figure out whether considering a restricted subset of JSONPath might produce a simpler solution while still enabling to address most mapping situations.

Implementing domain logic. In Chapter 4, we pointed out that the ability to implement domain logic should be considered in a generalized mapping language. The reason is that values stored in databases typically follow some formalism or syntax that may not be appropriate for an RDF representation. This feature was often overlooked in previous works, considering that it should be fulfilled by leveraging the underlying database query language. However, in a generalized context, we argue that it is fundamental to consider this feature at the level of the mapping language. We addressed this issue in xR2RML with mixed-syntax paths that can treat database values formatted in CSV/TSV, XML and JSON.

Yet, this feature does not cover all situations, as illustrated in section 8.4.1. Therefore, in section 5.8, we suggested a more generic way of implementing domain logic by means of custom functions, leveraging propositions of CSVW and R2RML-F. We believe that this is an interesting lead to make xR2RML even more generic and applicable to a broader set of use cases.

Such an extension should not be considered without evaluating carefully the impact in terms of SPARQL query rewriting. Indeed, in section 6.9 we illustrated that non-reversible custom functions may lead to situations where the rewriting of a SPARQL query is not possible. Thus, a custom function extension for xR2RML should provide the ability to define an inverse transformation function, similar to the R2RML `rr:inverseExpression` property.

Federation of distributed data sources. If we consider the broader picture, xR2RML and other languages (*e.g.* R2RML, RML, XSPARQL or CSVW) are meant to describe the mapping of one or only a few particular data sources to RDF. As such, they are suited to enable the development of wrappers for existing data sources. Nevertheless, for the Web of Data to meet the expectations of a Web-scale data integration, we need to pose queries that simultaneously span many data sources, create mashups so that novel applications can build upon such mashups to produce added value. Research on query federation engines [Schwarte et al., 2011; Görlitz & Staab, 2011; Corby et al., 2012; Macina et al., 2016; Montoya et al., 2013] and Triple Pattern Fragments [Verborgh et al., 2016] typically aim at this goal. Figure 22 is a hypothetical setup where SPARQL and Triple Pattern Fragment interfaces would be developed for various data sources, based on mapping descriptions that may be either xR2RML or any appropriate language. Note that DBpedia already provides both interfaces.

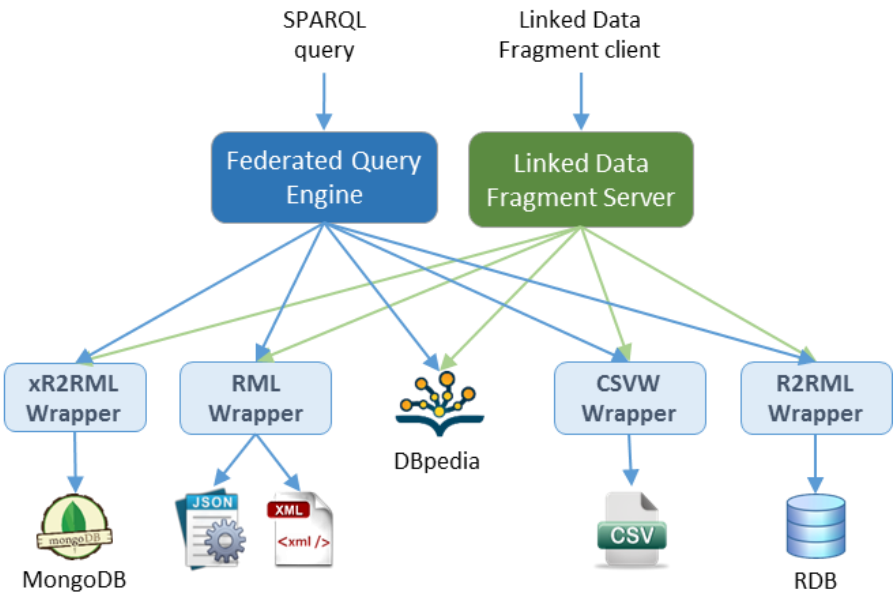


Figure 22: Hypothecial data integration scenario based on a federated query engine and a Triple Pattern Fragment server. Data source wrappers are based on various mapping languages.

***Schema mapping*.** Like with any mapping language, writing an xR2RML mapping involves expertise about the database schema and the domain ontologies that we consider. When either the database schema or the domain ontologies become large, manually writing the mappings becomes barely possible. To map large database schemas, it may be necessary to automate the production of an xR2RML direct mapping, at least as a starting point. Although the direct mapping was only formerly

defined in the context of RDBs [Arenas et al., 2012], it is easy to extend its principles beyond that scope. Inevitably, this shall raise the question of figuring out a schema for schema-less databases. As pointed out in section 3.3, TARQL [Cyganiak, 2013] and AstroGrid-D [Breitling, 2009] simply get rid of this issue by doing a direct mapping of each JSON document (respectively each XML document) independently of any schema consideration. Finally, if further semantic interoperability is required between the local ontology created by the direct mapping and domain ontologies, ontology alignment techniques should be considered [Euzenat & Shvaiko, 2013].

When it comes to large domain ontologies, it shall be useful to draw on existing techniques to automatically discover mappings between a database schema and domain ontologies [Shvaiko & Euzenat, 2005]. As an example, Karma [Knoblock et al., 2012] semi-automatically maps structured data sources to existing domain ontologies, and the authors suggest that the result could be exported as an R2RML mapping graph. Similarly, it should be possible to export the result as an xR2RML mapping graph.

The discussion above focuses on schema mapping, *i.e.* aligning entities of the database schema with classes and properties of the domain ontologies. Yet, for Web-scale data integration to give proper results, it shall also be necessary to draw links between resources that represent the same logical entities (this is generally called record linkage or entity matching). Many approaches have been proposed to address this topic such as [Tournaire et al., 2011]. Although this concern goes beyond our work in this thesis, it would be interesting to think about whether and how a data source mapped to RDF (with xR2RML or any other language) could cooperate in the matching of similar resources throughout the Web of Data.

Further Experimentations. In the future, we hope to pursue the work initiated in this thesis by applying it to concrete use cases. In particular, we hold contacts with biologists who conduct studies on the phenotype of different rice species [Le Ngoc et al., 2016]. Their data are collected in MongoDB instances, and we are engaging in a common reflection to align those data on existing domain ontologies. In this matter, xR2RML is being studied as a way to translate their data into RDF.

Towards more open data. Finally, one of the motivations of this thesis is to foster the publication of data as Linked Open Data on the Web, thus contributing to the building of worldwide knowledge commons. Beyond the scientific and technical challenges, among which we tried to address some of them, the most difficult obstacle may be the human factor. It occurs that many people and companies are reluctant to the idea of sharing their data without any other retribution than the attribution of the data to their creator/producer. The free software community had to (and still has to) grapple harshly to make the concept of free/open-source software seem natural to companies as well as developers. While open data seems now well understood and adopted within public institutions, there is still much work to be done within the private sector. This “evangelization” shall be achieved progressively along with success stories demonstrating the benefit of linked open data. In this matter, we, scientists, probably have a major role to play.

Appendix A. xR2RML Overview and Examples

This section provides an overview of the xR2RML mapping language along with examples illustrating the mapping of various types of databases and associated data formats.

An xR2RML mapping refers to logical sources to retrieve data from the input database. A logical source can be either an *xR2RML base table* (for input databases where tables or views exist), or an *xR2RML view* representing the result of executing a query against the input database. An xR2RML processor is provided with an xR2RML mapping description, a connection to the database and a reference formulation that specifies the syntax used to refer to data elements retrieved from query results: this can be a column name in the case of row-based systems (RDB, extensible column-store), a JSONPath expression in case of a NoSQL document store, an XPath expression in case of an XML native database, an LDAP attribute name in case of an LDAP directory, etc.

A logical source is mapped to RDF triples using one or more *triples maps*. As in R2RML, a triples map consists of a subject map that generates the subject of all RDF triples that will be generated from data elements, and multiple predicate-object maps that produce the predicate and object terms of RDF triples.

A.1 Mapping CSV data

The input database in the example below consists of one CSV document. We assume that the xR2RML processor is provided with a way to access that file, e.g. by means of a file descriptor or a URL (this is a choice of the xR2RML processor implementation). The reference formulation passed to the xR2RML processor instructs to use column names to refer to data elements.

As a CSV file simply consists of a single unnamed table, the logical source can simply be left empty.

Input data	`title, year, director` `Manhattan, 1979, Woody Allen` `Annie Hall, 1979, Woody Allen` `2046, 2004, Wong Kar-wai` `In the Mood for Love, 2000, Wong Kar-wai`
Mapping graph	`<#CSVTriplesMap>` `  rr:subjectMap [ rr:template "http://example.org/movie/{title}" ];` `  rr:predicateObjectMap [` `  rr:predicate ex:directedBy;` `  rr:objectMap [ xrr:reference "director" ];` `  ].`
RDF triples produced	`<http://example.org/movie/Manhattan>  ex:directedBy "Woody Allen".` `<http://example.org/movie/Annie%20Hall> ex:directedBy "Woody Allen".` `<http://example.org/movie/2046>  ex:directedBy "Wong Kar-wai".` `<http://example.org/movie/In%20the%20Mood%20for%20Love>` `  ex:directedBy "Wong Kar-wai".`

A.2 Mapping MongoDB JSON documents

The input database is a MongoDB database (NoSQL database storing JSON documents). The query in the logical source (property `xrr:query`) uses the proprietary JavaScript-based query language of MongoDB. It retrieves one JSON document from collection "movies", that lists movie directors and some movies they directed.

Without further instruction on how to parse the document, JSONPath expressions referring to data elements in the subject and object map will be evaluated against the whole document. For instance, a subject using expression `"$.directors.name"` will return two terms, while an object map using expression `"$.directors.movies.*"` will return four terms, two for each movie, whatever its director. This will result in mixing up directors and movies. To avoid this, the `rml:iterator` property in the logical source specifies that the triples map iteration should occur on each element of the array of directors.

References to data elements (`rr:template`, `xrr:reference`), as well as the iterator pattern, are expressed in JSONPath (i.e. the reference formulation passed to the xR2RML processor along with the database connection).

Input data	`{ "directors": [` `  { "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] },` `  { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }` `]}`
Mapping graph	`<#Directors>` `  xrr:logicalSource [` `  xrr:query "db.movies.find( { directors: { $exists: true} } )";` `  rml:iterator "$.directors.*";` `  ];` `  rr:subjectMap [ rr:template "http://example.org/director/{$.name}" ];` `  rr:predicateObjectMap [` `  rr:predicate ex:directed;` `  rr:objectMap [ xrr:reference "$.movies.*" ];` `  ].`
RDF triples produced	`<http://example.org/director/Woody%20Allen>` `  ex:directed "Manhattan", "Annie Hall".` `<http://example.org/director/Wong%20Kar-wai>` `  ex:directed "2046", "In the Mood for Love".`

A.3 Mapping XML data

The example below is very similar to the previous one, using an XML database supporting XQuery. The query in the logical source retrieves all "director" XML elements where the value of the “country” element is “China”. As in the previous example, to avoid mixing up directors and movies, an `rml:iterator` property is added to the logical source, specifying that the triples map iteration should occur on each "director" element.

References to data elements (`rr:template`, `xrr:reference`), as well as the iterator pattern, use the XPath syntax.

Input data	`<directors>` `  <director name="Wong Kar-wai">` `  <country>China</country>` `  <movies>` `  <movie>2046</movie>` `  <movie>In the Mood for Love</movie>` `  </movies>` `  </director>` `  <director name="Woody Allen">` `  <country>USA</country>` `  <movies>` `  <movie>Manhattan</movie>` `  <movie>Annie Hall</movie>` `  </movies>` `  </director>` `</directors>`
Mapping graph	`<#Directors>` `  xrr:logicalSource [` `  xrr:query """for $i in //directors/director` `  where $i/country eq "China"` `  return $i """;` `  rml:iterator "//director";` `  ];` `  rr:subjectMap [` `  rr:template "http://example.org/director/{/director/@name}";` `  ];` `  rr:predicateObjectMap [` `  rr:predicate ex:directed;` `  rr:objectMap [ xrr:reference "//movie" ];` `].
RDF triples produced	`<http://example.org/director/Wong%20Kar-wai>` `  ex:directed "2046", "In the Mood for Love".`

A.4 Mapping data with mixed formats

In some use cases, values are stored in a format that is not the native database format. For instance, an application designer may choose to embed JSON, CSV, or XML values in the cells of a relational database, for performance concerns or application design constraints.

xR2RML can reference data elements within such mixed contents with *mixed-syntax paths*. A path with mixed-syntax consists of the concatenation of one or several slash-separated path expressions. Each individual path is enclosed in a syntax path constructor: `Column(column-name)`, `CSV(column-name)`, `TSV(column-name)`, `JSONPath(JSONPath-expression)`, `XPath(XPath-expression)`.

In the example below, the logical source is a relational table with two columns. The second column, Movies, holds values formatted as JSON arrays. The `xrr:reference` property of the object map uses the mixed-syntax path "`Column(Movies)/JSONPath($.*)`" expression that selects values from column "Movies" and evaluates JSONPath expression "`$.*`" against the values.

<table>
<tr><td>Input data</td><td>Table DIRECTORS:
<table><tr><td>Name</td><td>Movies</td></tr><tr><td>Wong Kar-wai</td><td>["2046", "In the Mood for Love"]</td></tr><tr><td>Woody Allen</td><td>["Manhattan", "Annie Hall"]</td></tr></table></td></tr>
<tr><td>Mapping graph</td><td><pre><#Directors>
    rr:logicalTable [
       rr:tableName "DIRECTORS";
    ];
    rr:subjectMap [rr:template "http://example.org/director/{Name}"];
    rr:predicateObjectMap [
       rr:predicate ex:directed;
       rr:objectMap [xrr:reference "Column(Movies)/JSONPath($.*)"];
    ].</pre></td></tr>
<tr><td>RDF triples produced</td><td><pre><http://example.org/director/Woody%20Allen>
          ex:directed "Manhattan", "Annie Hall".
<http://example.org/director/Wong%20Kar-wai>
          ex:directed "2046", "In the Mood for Love".</pre></td></tr>
</table>

A.5 Generating an RDF collection from a list of values

As illustrated in the previous example, several RDF terms can be generated by a term map during one triples map iteration step. While this can lead to the generation of several triples, this can also be used to generate hierarchical values in the form of RDF collections or containers.

The example below was already presented in section A.2. In the object map we add an `rr:termType` property with value `xrr:RdfList`. All RDF terms produced by the object map during one triples map iteration step are now grouped as members of one term of type `rdf:List` (denoted as "(a b c…)" in Turtle syntax).

Additionally, assume we want to add a language tag to the movie titles. The object map describes the generation of RDF lists, hence it would not make sense to add an `rr:language` property. To state properties about the members of the generated RDF list, we need a *nested term map*, introduced by the `xrr:nestedTermMap` property. A nested term map accepts the same properties as a term map, but it applies to members of RDF collection/container terms generated by its parent term map.

<table>
<tr><td>Input data</td><td><pre>{ "directors": [
  { "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] },
  { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }</pre></td></tr>
</table>

	]}
Mapping graph	<#Directors> xrr:logicalSource [xrr:query "db.movies.find({ directors: { $exists: true} })"; rml:iterator "$.directors.*";]; rr:subjectMap [rr:template "http://example.org/director/{$.name}"]; rr:predicateObjectMap [rr:predicate ex:directed; rr:objectMap [xrr:reference "$.movies.*"; rr:termType xrr:RdfList; xrr:nestedTermMap [rr:language "en"]];].
RDF triples produced	<http://example.org/director/Woody%20Allen> ex:directed ("Manhattan"@en "Annie Hall"@en). <http://example.org/director/Wong%20Kar-wai> ex:directed ("2046"@en "In the Mood for Love"@en).

A.6 Generating an RDF container from a cross-reference

The example below uses a referencing object map to describe a cross-reference relationship between logical resources. In addition, it generates an RDF bag from the result of the join condition in the referencing object map.

Triples map `<#Movies>` generates IRIs for the movies. The referencing object map in triples map `<#Directors>` uses IRI generated in triples map `<#Movies>` as the members of an RDF bag (property `rr:term:Type xrr:RdfBag`).

The join condition in triples map `<#Directors>` produces a result if a movie title (`rr:parent "$.title"`) matches at least one movie in the list of movies associated with each director (`rr:child "$.movies.*"`).

Input data	{ "directors": [{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] }, { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }]} { "movies": [{ "title": "Manhattan", "year": "1979" }, { "title": "Annie Hall", "year": "1977" }, { "title": "2046", "year": "2004" }, { "title": "In the Mood for Love", year: "2000"}]}
Mapping graph	<#Movies> xrr:logicalSource [xrr:query "db.movies.find({ movies: { $exists: true} })"; rml:iterator "$.movies.*";

	``]; rr:subjectMap [rr:template "http://example.org/movies/{$.title}"]. <#Directors> xrr:logicalSource [xrr:query "db.movies.find({ directors: { $exists: true} })"; rml:iterator "$.directors.*";]; rr:subjectMap [rr:template "http://example.org/director/{$.name}"]; rr:predicateObjectMap [rr:predicate ex:directed; rr:objectMap [rr:parentTriplesMap <#Movies>; rr:joinCondition [rr:child "$.movies.*"; rr:parent "$.title";]; rr:termType xrr:RdfBag;];]. ```
Generated RDF triples	``` <http://example.org/director/Woody%20Allen> ex:directed [a rdf:Bag; rdf:_1 <http://example.org/movie/Manhattan>"; rdf:_1 <http://example.org/movie/Annie%20Hall>]. <http://example.org/director/<Wong%20Kar-wai> ex:directed [a rdf:Bag; rdf:_1 <http://example.org/movie/2046>"; rdf:_2 <http://example.org/movie/In%20the%20Mood%20for%20Love>]. ```

Appendix B. The xR2RML Mapping Language Specification

B.1 Preliminary definitions

B.1.1 xR2RML mapping graphs and mapping documents

An **xR2RML mapping** defines a mapping from a database to RDF; it is represented as an RDF graph called an **xR2RML mapping graph**.

An **xR2RML mapping document** is any document written in the Turtle RDF syntax that encodes an **xR2RML mapping graph**.

Any R2RML mapping graph is a valid **xR2RML mapping graph** (backward compatibility).

B.1.2 xR2RML processor

An **xR2RML processor,** or **processing engine,** is a system that, given an xR2RML mapping and an input database, provides access to the output RDF dataset.

An **xR2RML processor** has access to an execution environment consisting of:

- An **xR2RML mapping document**, as defined above.
- A **connection** to the input database, used by the xR2RML processor to evaluate queries against the input database. It must be established with sufficient privileges for read access to all database elements (tables, views, documents, objects…) that are referenced in the xR2RML mapping.
- A **reference formulation** (optional): this concept, introduced by RML, names a syntax used to reference data elements within results of a query run against the database connection. In conformance with requirement 1 (§4.2.1), the reference formulation is not mentioned in the mapping language, but is typically provided as configuration parameter. If it is not provided, it defaults to “column name” in order to ensure backward compatibility with R2RML.
- A **query language identifier** (optional) identifies which query language shall be used to query the database, in case several languages are supported.
- A **base IRI** used in resolving relative IRIs produced by the xR2RML mapping (optional).

It is the responsibility of an xR2RML processor developer to document how to provide the processor with this context information.

B.2 Triples Maps and Logical Sources

B.2.1 xR2RML Triples Map

An *xR2RML triples map* specifies rules for translating data elements of a logical source to zero or more RDF triples. The RDF triples generated from one data element (row, document, set of XML elements, etc.) in the logical source all share the same subject.

An xR2RML triples map extends R2RML triples map by referencing a logical source instead of a logical table. An xR2RML triples map is represented by a resource that references the following resources:

- It must have exactly one **`xrr:logicalSource`** property. Its object is a logical source that specifies a table or a query result to be mapped to triples.
- It must have exactly one subject map that specifies how to generate a subject for each data element of the logical source (row, document, set of XML elements, etc.). A subject map may be specified in two ways:
  - using the `rr:subjectMap` property, whose value must be the subject map, or
  - using the constant shortcut property `rr:subject`.
- It may have zero or more `rr:predicateObjectMap` properties, whose values must be predicate-object maps. They specify pairs of predicate maps and object maps that, together with the subjects generated by the subject map, may form one or more RDF triples for each data element.

B.2.2 Defining a Logical Source

***Definition 18**. An **xR2RML logical source** (property **`xrr:logicalSource`**) extends the R2RML concept of logical table (property `rr:logicalTable`). A logical source is the result of running a query against to the input connection. A logical source is either an **xR2RML base table** or an **xR2RML view**.*

*An **xR2RML base table** is a logical source containing data from a table in the input database. It simply extends the concept of R2RML table or view to the context of tabular databases beyond relational databases (e.g. CSV, extensible column store). An xR2RML base table is represented by a resource that has exactly one `rr:tableName` property. Its object is a string literal representing the table name.*

*An **xR2RML view** is a logical source whose content is the result of executing a query against the input database. It is represented by a resource that has exactly one `xrr:query` property. Property **`xrr:query`** extends RML property `rml:query`. The object of property `xrr:query` is a string literal representing a valid expression with regards to the query language supported by the input database.*

*A logical source may have one data property **`rml:iterator`** that specifies the iteration pattern to apply to data retrieved from the input database (section B.2.3). Its object is an expression written*

using the reference formulation. The `rml:iterator` property is ignored in the context of tabular result sets in which data is accessed by column names.

*An `rml:iterator` property may be complemented with any number of **`xrr:pushDown`** properties (section B.2.3) that specify how some values must be added within the documents that result of the `rml:iterator`. The object of a `xrr:pushDown` property is an instance of the **`xrr:PushDown`** class, that must have exactly one xrr:reference property and one **`xrr:as`** property.*

*A logical source may have any number of properties **`xrr:uniqueRef`** that specify a unique data element reference within the documents retrieved by the xrr:query property. This property shall be used during the query optimization when rewriting a SPARQL query into the target database query language. The unique data element reference is an expression written according to the syntax specified by the reference formulation.*

Note that xR2RML logical source and R2RML logical table definitions may equally be used in the context of a relational database. Examples:

R2RML logical table	Equivalent xR2RML logical source
`[] rr:logicalTable [` `  rr:tableName "SOME_TABLE".` `]`	`[] xrr:logicalSource [` `  rr:tableName "SOME_TABLE";` `]`
`[] rr:logicalTable [` `  rr:sqlQuery` `  "SELECT NAME, DATE FROM MOVIES".` `]`	`[] xrr:logicalSource [` `  xrr:query` `  "SELECT NAME, DATE FROM MOVIES".` `]`

The table below shows examples of xR2RML logical source definition with different flavors of input databases.

Type of database	Logical source definition
Relational database	`[] rr:logicalTable [` `  rr:sqlQuery """` `  SELECT TITLE FROM MOVIES` `  WHERE YEAR > 1980` `  ORDER BY YEAR` `  LIMIT 10""";` `];`
XML file. The xR2RML processor is provided with a file URL, e.g. http://example.org/movies.xml, and the XPath reference formulation. Therefore no further query is required (no xrr:query property). An iterator defines the pattern of XML elements to iterate on. XPath is used to refer to data elements within the XML data returned by the database.	`[] xrr:logicalSource [` `  rml:iterator "//movie";` `];`
REST-based Web service returning an XML stream based on parameters passed in the HTTP GET query	`[] xrr:logicalSource [` `  xrr:query "?minYear=1980&limit=10";`

string. The service URL (e.g. http://example.org/service) and XPath reference formulation are passed to the xR2RML processor by configuration, while the HTTP query string is provided by the xrr:query property.	`rml:iterator "//movie";` `];`
XML database supporting XQuery. XPath is used to describe the iterator and later on to refer to data elements within the XML data returned by the database.	`[] xrr:logicalSource [` `xrr:query """for $i in //movies/movie` `where $i/year gt 1980` `order by $i/@title` `return $i/@title""";` `rml:iterator "//movie";` `];`
JSON file. The xR2RML processor is provided with file URL and the JSONPath reference formulation. No further query is required (no xrr:query property). An iterator defines the pattern to iterate on.	`[] xrr:logicalSource [` `rml:iterator "$.movies.*";` `];`
MongoDB database (document store). MongoDB Shell Query Language is used to search for documents in collection "movies". Then, JSONPath is used to refer to data elements within the JSON documents returned by the database.	`[] xrr:logicalSource [` `xrr:query '''` `db.movies.find({"year":{$gt: 1980}})` `''';` `xrr:uniqueRef "$.movieId"` `];`
Cassandra (extensible column store) using Cassandra Query Language (CQL). Data element are referenced by column name (reference formulation passed to the xR2RML processor).	`[] xrr:logicalSource [` `xrr:query """` `SELECT TITLE, YEAR FROM Movies` `LIMIT 10` `""";` `];`
AllegroGraph (RDF graph store) using SPARQL. The column name reference formulation is applied to a SPARQL result set: the result set is considered as a table in which columns are named after variable names.	`[] xrr:logicalSource [` `xrr:query """` `select ?title ?year` `where { ?movie a ex:Movie;` `ex:name ? title;` `ex:year ?year. }` `filter (?year > "1980"^^xsd:integer)` `order by ?year` `limit 10""";` `];`

B.2.3 xR2RML Triples Map Iteration Model

In R2RML, the row-based iteration implicitly occurs on a set of rows read from a logical table. xR2RML applies this principle to other row-based systems such as CSV/TSV files and extensible column stores, but also to any tabular result such as a SPARQL result sets. In the context of non-row-based databases, the model is extended to a **document-based iteration model**: a document is one result of a result set, whatever the form of such a result. Typically, the document-based iteration applies to a set of JSON documents retrieved from a NoSQL document store, or a set of XML documents retrieved from an XML

database. In the case of data sources that do not natively provide iterators over results, for instance a simple file or a Web service returning an XML stream at once, a single iteration occurs on the whole document.

The document-based iteration model alone may not be sufficient to fulfill any iteration need. This is particularly true for hierarchical data models such as JSON and XML. Therefore, the iteration can be modified using the RML iterator concept. xR2RML amends the definition of a logical source in Definition 19.

> ***Definition 19**. An **iterator** is defined within an xR2RML logical source by means of the `rml:iterator` property. It specifies the iteration pattern to apply to data retrieved from the input database. The object of an `rml:iterator` property is a valid path expression written using the syntax specified by the reference formulation.*

Let us consider the JSON example document below that describes movie directors and movies:

```
{ "type": 3,
 "directors": [ { "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] },
    { "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }
]}
```

We consider the following xR2RML following mapping graph:

```
<#Directors>
    xrr:logicalSource [ xrr:query "db.movies.find( { directors: { $exists: true} } )" ];
    rr:subjectMap [ rr:template "http://example.org/director/{$.name}"; ];
    rr:predicateObjectMap [
       rr:predicate ex:directed;
       rr:objectMap [ xrr:reference "$.directors.*.movies.*"; ];
    ].
```

In this mapping, the subject map returns two terms (one per director) while the object map returns four terms (one per movie in the document). Consequently, the generated triples mix up all directors and movies:

```
<http://example.org/director/Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/Woody%20Allen> ex:directed "2046".
<http://example.org/director/Woody%20Allen> ex:directed "In the Mood for Love".
<http://example.org/director/Wong%20Kar-wai> ex:directed "Manhattan".
<http://example.org/director/Wong%20Kar-wai> ex:directed "Annie Hall".
<http://example.org/director/Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".
```

To deal with such cases, xR2RML relies on the concept of iterator defined in RML. With the `rml:iterator` property, the previous example is modified as shown below:

```
<#Directors>
    xrr:logicalSource [
```

```
        xrr:query "db.movies.find( { directors: { $exists: true} } )";
        rml:iterator "$.directors.*";
    ];
    rr:subjectMap [ rr:template "http://example.org/director/{$.name}"; ];
    rr:predicateObjectMap [
        rr:predicate ex:directed;
        rr:objectMap [ xrr:reference "$.movies.*"; ];
    ].
```

The `rml:iterator` property indicates that, within the document retrieved, the triples map iteration should be performed on each director element rather than on the whole document. The iterator now yields two separate documents:

```
{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"] }
{ "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"] }
```

In turn, the first document yields the first two triples below, while the second document yields the 3rd and 4th triples:

```
<http://example.org/director/Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/Wong%20Kar-wai> ex:directed "In the Mood for Love".
```

Let us now assume that we would like field “type” (whose value is 3, whatever this means) to be part of the subject terms, e.g. <http://example.org/director/**3-**Woody%20Allen>. The documents yielded by the iterator do not contain field “type” since it is defined higher in the document hierarchy. To address this issue, we define an element of class xrr:PushDown with the xrr:pushDown property:

> An `xrr:PushDown` **instance** is defined within an xR2RML logical source by means of the `xrr:pushDown` property. It is valid only as a companion of an `rml:iterator` property. It specifies a reference (property `xrr:reference`), written using the reference formulation (section **Erreur ! Source du renvoi introuvable.**), whose value must be added in the documents yielded by the iterator as a new data element whose name is given by the `xrr:as` property of the `xrr:PushDown` instance.

In our example, this should look like:

```
<#Directors>
    xrr:logicalSource [
        xrr:query "db.movies.find( { directors: { $exists: true} } )";
        rml:iterator "$.directors.*";
        xrr:pushdown [ xrr:reference "$.type" xrr:as "newType" ];
    ];
    rr:subjectMap [
        rr:template "http://example.org/director/{$.newType}-{$.name}";
    ];
    rr:predicateObjectMap [
        rr:predicate ex:directed;
```

```
        rr:objectMap [ xrr:reference "$.movies.*"; ];
    ].
```

As a result, a field "newType" (property `xrr:as`) is created within the documents yielded by the iterator. The value of this field is the result of evaluating the "$.type" reference (xrr:reference property) against the documents fetched from the database. Example:

```
{ "name": "Wong Kar-wai", "movies": ["2046", "In the Mood for Love"], "newType": 3 }
{ "name": "Woody Allen", "movies": ["Manhattan", "Annie Hall"], "newType": 3 }
```

Finally, the following triples are produced:

```
<http://example.org/director/3-Woody%20Allen> ex:directed "Manhattan".
<http://example.org/director/3-Woody%20Allen> ex:directed "Annie Hall".
<http://example.org/director/3-Wong%20Kar-wai> ex:directed "2046".
<http://example.org/director/3-Wong%20Kar-wai> ex:directed "In the Mood for Love".
```

B.3 Creating RDF terms with Term Maps

B.3.1 xR2RML Term Maps

R2RML defines a **term map** as a function that generates **RDF terms** from a logical table row. A term map may be a subject map, predicate map, object map or graph map. A term map must be exactly one of the following types:

- a constant-valued term map (property rr:constant)
- a column-valued term map (property rr:column)
- a template-valued term map (property rr:template).

R2RML treats all values from the input database as literals expressed in built-in data types. To deal with structured values such as collections or key-value associations, xR2RML term maps extend R2RML term maps such that structured values can be parsed, and data elements within structured values can be selected to build RDF terms. xR2RML's extensions are described in this section.

B.3.1.1 Constant-, Column-, Reference- and Template-valued Term Maps

R2RML properties `rr:column` and `rr:template` reference columns of a logical table. xR2RML not only references columns but also any data element within structured values. xR2RML relies on the `rml:reference` property that extends property `rr:column`. Its object is an expression using the syntax mentioned in the reference formulation. Furthermore, xR2RML introduces the possibility to reference data elements in data with mixed formats (§B.3.2.2). Thus, xR2RML extends property `rml:reference` with property `xrr:reference`. This leads to the following definition of an xR2RML term map. xR2RML changes to R2RML are highlighted.

***Definition 20**. A constant-valued term map is represented by a resource that has exactly one `rr:constant` property. A constant-valued term map always generates the same RDF term.*

A column-valued term map has exactly one `rr:column` property. The value of the `rr:column` property is a valid column name.

A reference-valued term map has exactly one xrr:reference property. The value of the xrr:reference property is a valid reference to a data element, expressed using the syntax defined by the reference formulation.). A reference-valued term map term map may have any number of xrr:pushDown properties (section B.3.2.4) if and only if it has an xrr:nestedTermMap property.

A template-valued term map has exactly one `rr:template` property. The value of the `rr:template` property is a valid string template. A string template is a format string used to build strings from multiple components. It uses valid references to data elements by enclosing them in curly braces ("{"and "}"). Each reference is expressed using the syntax defined by the reference formulation.

B.3.1.2 Term Types of Term Maps

RDF terms generated by a term map have a term type (`rr:termType`) that may be one of the three R2RML term types: `rr:Literal`, `rr:BlankNode` or `rr:IRI`.

***Definition 21**. xR2RML extends the rr:termType property with four new values, hereafter referred to as **RDF collection or container term types**:*

- *A term map with **`xrr:RdfList`** as value of property `rr:termType` generates an RDF collection of type rdf:List;*
- *A term map with **`xrr:RdfSeq`** as value of property `rr:termType` generates an RDF container of type rdf:Seq;*
- *A term map with **`xrr:RdfBag`** as value of property `rr:termType` generates an RDF container of type rdf:Bag;*
- *A term map with **`xrr:RdfAlt`** as value of property `rr:termType` generates an RDF container of type rdf:Alt.*

B.3.1.3 Nested Term Maps

Hierarchical data such as JSON or XML documents conventionally have more than one level of nesting, resulting in hierarchical values that may need to be parsed recursively to nest RDF collections and containers, e.g. to build an RDF collection of RDF collections.

***Definition 22**. An xR2RML term map may have an **xrr:nestedTermMap** property, whose range is the **xrr:NestedTermMap** class.*

In a column-valued or reference-valued term map, the `xrr:nestedTermMap` property describes how to translate values produced by the `rr:column` or `xrr:reference` properties into RDF terms.

In a template-valued term map, the `xrr:nestedTermMap` property describes how to translate values produced by applying the template string to input values into RDF terms.

In a constant-valued term map, it is invalid to define a nested term map.

***Definition 23.** A **nested term map** may have the properties below:*

- ***xrr:reference** bears the same semantics as in the context of a term map. Its object is a valid path expression (possibly a mixed-syntax path, see §B.3.2.2). Evaluation of the path expression is performed against values retrieved by the parent of the nested term map. This parent may be a term map or a nested term map.*
- ***rr:template** bears the same semantics as in the context of a term map. References enclosed in capturing curly braces are valid path expressions (possibly mixed-syntax paths), they apply to values retrieved in the parent of the nested term map. This parent may be a term map or a nested term map.*
- ***xrr:nestedTermMap** is used to recursively parse any depth of nested structured values;*
- ***xrr:pushDown** properties if and only if there exists an `xrr:nestedTermMap` property (see section B.3.2.4).*
- ***rr:termType** bears the same semantics as in the context of a term map;*
- ***rr:language** bears the same semantics as defined in R2RML;*
- ***rr:datatype** bears the same semantics as defined in R2RML.*

*A **simple nested term map** is a nested term map that has no `xrr:reference` nor rr:template property. A simple nested term map is used to qualify terms of an RDF collection or container generated by its parent term map or nested term map, i.e. assign them an optional term type, data type or language tag.*

*A **reference-valued nested term map** is a nested term map that has exactly one `xrr:reference` property.*

*A **template-valued nested term map** is a nested term map that has exactly one `rr:template` property.*

xrr:nestedTermMap vs. rr:termType:

A nested term map *N* describes how to translate into RDF terms values produced by its parent *P*, *P* may be a term map or a nested term map.

If *P* has no `rr:termType` property, it simply returns values produced by *N*, therefore the term type of P is that of *N*.

If *P* has an rr:termType property with an RDF collection or container term type as object, then values produced by *N* will be assembled in an RDF collection or container.

Lastly, *P* should not have an `rr:termType` property with an R2RML term type (literal, blank node, IRI) or in other words, a nested term map cannot be used in the context of a term map or nested term map that has an R2RML term type (literal, IRI, blank node). Thus:

> ***Definition 24****. If a term map or nested term map has an `xrr:nestedTermMap` property, then it should have either no `rr:termType` property or an `rr:termType` property with an RDF collection or container term type. Formally:*
>
> *?P is an rr:TermMap or xrr:NestedTermMap.*
>
> *?P xrr:nestedTermMap ?N.*
>
> *?P rr:termType ?tt.*
>
> *⇒ ?tt is one of `xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt`*
>
> *A term map or nested term map with an RDF collection or container term type and no xrr:nestedTermMap property is assumed to have a **default `xrr:nestedTermMap` property** defined as follows:*
>
> *- If the parent term map or nested term map is reference-valued:*
>
> *`xrr:nestedTermMap [ rr:termType rr:Literal ];`*
>
> *- If the parent term map or nested term map is template-valued:*
>
> *`xrr:nestedTermMap [ rr:termType rr:IRI ];`*

Finally, as defined in R2RML, properties `rr:language` and `rr:datatype` apply when generating literals only:

> ***Definition 25****. A term map or nested term map may have an `rr:language` or `rr:datatype` property only if its term type is `rr:Literal` (either stated by property `rr:termType` or inferred as a default value).*

Nested term maps are exemplified in section B.3.3.

B.3.2 Referencing data elements

B.3.2.1 Referencing simple data elements

The table below exemplifies the use of properties `rr:column`, `xrr:reference` and `rr:template` to reference simple data elements (i.e. with non-mixed data formats) from the logical source.

Logical source	Term map
Relational database: either `rr:column` or `xrr:reference` can be used to name a column.	`[] rr:column "NAME".` `[] xrr:reference "NAME".` `[] rr:template "{NAME}".`

Extensible column store: properties `xrr:reference` and `rr:template` reference data elements by column names.	`[] xrr:reference "NAME".` `[] rr:template "{NAME}".`
XML database supporting: properties `xrr:reference` and `rr:template` reference data elements by XPath expressions.	`[] xrr:reference "//name".` `[] rr:template "{//name }".`
NoSQL document store: `xrr:reference` and `rr:template` reference data elements using a valid JSONPath expression.	`[] xrr:reference "$.name".` `[] rr:template "{$.name }".`
RDF graph store accessed using a SPARQL SELECT query: `xrr:reference` and `rr:template` reference data elements by name of variable returned in the SPARQL result set.	`[] xrr:reference "?name".` `[] rr:template "{?name }".`

Remark: If a term map references a structured value but does not parse it using a nested term map, then generated RDF terms are string literals containing a simple serialization of structured values. Example:

Input data	`{ "person": { "FirstName":"John", "LastName":"Smith" } }`
Term map	`[] rr:objectMap [` `  xrr:reference "$.person";` `];`
Generated RDF term	The structured value matching the JSONPath expression "$.person" is returned as a string literal in quotes: `'{ "FirstName":"John", "LastName":"Smith" }'`

B.3.2.2 Referencing data elements with mixed data formats

In some use cases, databases are commonly used to store values written in a data format that they cannot interpret. For instance, an application designer may choose to embed JSON, CSV, or XML values in the cells of a relational table, for performance concerns or application design constraints.

To reference data elements within such mixed contents, xR2RML allows a term map to reference data elements with **mixed-syntax paths**:

> ***Definition 26***. *Properties `xrr:reference` and `rr:template` may use **mixed-syntax paths** to reference data elements across data in different formats. A mixed-syntax path consists of the concatenation of several path expressions separated by the slash '/' character. Each individual path is enclosed in a **syntax path constructor** naming the path syntax explicitly. Existing constructors are:*
>
> - ***Column**(column-name): applies to row/column databases such as relational database and extensible column-store.*
> - ***CSV**(column-name), **TSV**(column-name): applies to data formatted as comma-separated or tab-separated values. Column-name may be a 0-based column index, or an actual column name if a head line provides column names.*
> - ***JSONPath**(JSONPath-expression): applies to any data formatted in JSON.*

- ***XPath****(XPath-expression): applies to any data formatted in XML.*

In expressions enclosed in a syntax path constructor, special characters '/', '(', ')', '{' and '}' must be escaped with a '\'. Since, in Turtle syntax, the '\' character itself must be escaped with an additional '\', special characters are escaped with '\\'.

Example:

Input data	Relational table with one column: `Name` `{ "FirstName":"John", "LastName":"Smith" }`
Logical source definition and Term map	`[] xrr:logicalSource [ … ];` `...` `rr:objectMap [` `xrr:reference "Column(Name)/JSONPath($.FirstName)";` `rr:language "en";` `];`
Generated RDF term	`"John"@en`

From the example above, it can be noticed that (i) the leftmost syntax path constructor (Column) is consistent with the reference formulation, and (ii) data elements referenced by mixed-syntax path `"Column(Name)/JSONPath($.FirstName)"` are formatted in JSON, corresponding to the rightmost syntax path constructor. More generally:

Definition 27*. The leftmost syntax path constructor of a mixed-syntax path must be consistent with the reference formulation.*

- *Constructors Column(), CSV() and TSV() apply with the column name reference formulation ,*
- *Constructor XPath() applies with the XPath reference formulation,*
- *Constructor JSONPath() applies with the JSONPath reference formulation.*

The format of data retrieved by a mixed-syntax path is the format of the rightmost syntax path constructor.

B.3.2.3 Production of multiple RDF terms

In a row-based logical source, a column reference returns exactly one scalar value per triples map iteration step: the value of the cell identified by "column name" in the current row. Thus, an R2RML term map generates zero or one RDF term during each iteration step, ultimately a triples map generates zero or one triple during each iteration step.

Due to the tree-like nature or JSON and XML data formats, JSONPath and XPath expressions allow addressing not only literals but also structured values. Thus, using the `xrr:reference` or rr:template properties with a JSONPath or XPath expression may return multiple values during each triples map iteration step. Hence the introduction of the term map iteration.

> ***Definition 28**. A **term map iteration** is a process that occurs in a term map during each triples-map iteration step. Thus, a reference-valued or template-valued term map generates zero to any number of RDF terms during each triples-map iteration step.*

Examples:

Input data retrieved in one triples-map iteration step	`{` `  "FirstNames":` `  ["John", "Albert"],` `  "LastName": "Smith"` `}`	`<person>` `  <FirstNames>` `  <item>John</item>` `  <item>Albert</item>` `  </FirstNames>` `  <LastName>Smith</LastName>` `</person>`
Term map	`[] rr:objectMap [` `  xrr:reference` `  "$.FirstNames.*";` `  ];`	`[] rr:objectMap [` `  xrr:reference` `  "/person/FirstNames/item";` `  ];`
Generated RDF terms	`"John"` `"Albert"`	`"John"` `"Albert"`

The term map iteration applies identically in the context of mixed-syntax paths. Example:

Input data	`<person>` `  <name>John Smith</name>` `  <items>[1,2,3]</items>` `</person>` XML element "items" contains a value expressed as a JSON array.
Term map	`[] xrr:logicalSource [ ... ]` `  …` `  rr:objectMap [` `  xrr:reference "XPath(\\/person\\/items)/JSONPath($.*)";` `  rr:datatype xsd:integer;` `  ]` The last expression of the mixed-syntax path, "JSONPath($.*)", indicates that (i) value "[1,2,3]" is formatted in JSON syntax, and (ii) it must be parsed as such using the "$.*" JSONPath expression.
Generated RDF terms	`1^^xsd:integer` `2^^xsd:integer` `3^^xsd:integer`

A template-valued term map may reference several data elements from the logical source, captured by curly braces ('{' and '}'). If at least one of the data elements referenced in a template string produces several terms, then the following applies:

> ***Definition 29***. *A template-valued term map produces RDF terms by performing a Cartesian product between all values produced by all data elements referenced in the template.*

Example:

Input data	`{` `  "FirstNames": '["John", "Albert"]',` `  "LastName": "Smith"` `}`
Term map	`[] xrr:logicalSource [ … ];` `  rr:subjectMap [` `  rr:template "http://example.org/{$.FirstNames.*}/{$.LastName}";` `  ] ;`
Generated RDF terms	The template performs a Cartesian product between "Smith" and ["John", "Albert"], resulting in two terms: <http://example.org/John/Smith> <http://example.org/Albert/Smith>

Finally, below we define the behavior of a triples map in which one or several term maps generate multiple RDF terms during a single triples map iteration step:

> ***Definition 30***. *During each iteration of an xR2RML triples map, triples are generated as the Cartesian product between RDF terms produced by the subject map and each predicate-object map. Predicate-object couples result of the Cartesian product between RDF terms produced by each predicate and object map.*
>
> *Note that a graph map may also produce multiple terms, in which case triples are produced simultaneously in several target graphs.*

xR2RML vs. RML: RML makes the restriction that a subject map should return zero or one value during each triples map iteration. In the case of xR2RML, we make no such restriction so that the Cartesian product be possibly applied between multiple subjects, multiple predicate-object couples, and multiple graph IRIs. Besides, RML does not describe how a template with several multi-valued references is processed. xR2RML states that the Cartesian product applies equally in this case, whether the template is used as a subject, predicate, object or graph map.

In the example below, during one triples map iteration step, the subject map produces two RDF terms <http://example.org/company/Dell> and <http://example.org/company/Asus>, while the object map produces two literals "Laptop" and "Desktop". A Cartesian product between the two subjects and the two objects results in the production of four triples:

<table>
<tr><td>Input data: one row retrieved from a relational table, values are formatted in JSON in column “cos”, and XML in column “products”</td><td>
<table>
<tr><td>cos</td><td>products</td></tr>
<tr><td>["Dell", "Asus"]</td><td><list>
  <product>Laptop</product>
  <product>Desktop</product>
</list></td></tr>
</table>
</td></tr>
<tr><td>Mapping graph</td><td>[] xrr:logicalSource [...];
  rr:subjectMap [
  rr:template "http://example.org/{Column(cos)/JSONPath($.*)}";
  ];
  rr:predicateObjectMap [
  rr:predicate ex:produces;
  rr:objectMap [
  xrr:reference "Column(products)/XPath(\\/list\\/*)";
  ];
  ];</td></tr>
<tr><td>Generated triples</td><td><http://example.org/Dell> ex:produces "Laptop".
<http://example.org/Dell> ex:produces "Desktop".
<http://example.org/Asus> ex:produces "Laptop".
<http://example.org/Asus> ex:produces "Desktop".</td></tr>
</table>

B.3.2.4 Pushing down data elements during a term map iteration

As seen in the previous section, a **term map iteration** may occur in the context of hierarchical data formats. Within a term map iteration, it may be needed to access data elements higher in the hierarchical documents, such as JSON fields that are outside of the iteration and thus no longer available. To deal with such cases, a **reference-valued term map** or **a reference-valued nested term map** may have any number of xrr:pushDown properties, whose semantics is that defined in the context of a logical source (section B.2.3).

Example:

<table>
<tr><td>Input data</td><td>{ "id": 5,
  "FirstNames": '["John", "Albert"]',
  "LastName": "Smith"
}</td></tr>
<tr><td>Term map</td><td> rr:objectMap [
  rr:reference " $.FirstNames";
  xrr:pushDown [xrr:reference "$.id"; xrr:as "personId"];
  xrr:nestedTermMap [
  rr:template "Person {$.personId}: {$.*}";
  rr:termType xrr:Literal;
  ]
  ] ;</td></tr>
</table>

Generated RDF terms	"Person 5: John" "Person 5: Albert"

B.3.2.5 Production of RDF collections or containers

A term map with an RDF collection or container term type generates one RDF term during each triples map iteration step. The elements of the collection or container are the RDF terms produced by the term map, whether using property `rr:column`, `xrr:reference` or `rr:template`.

In the example below, the triples map generates one triple per iteration step, the object of this triple is an RDF bag (itself consisting of several triples):

Input data: JSON document retrieved in a single iteration step	`<company "name"="Dell">` `  <products>` `  <product>Laptop</product>` `  <product>Desktop</product>` `  </products>` `</company>`
Mapping graph	`[] xrr:logicalSource [ ... ];` `  rr:subjectMap [` `  rr:template "http://example.org/{/company/@name}";` `  ];` `  rr:predicateObjectMap [` `  rr:predicate ex:builds;` `  rr:objectMap [` `  xrr:reference "//company/products/*";` `  rr:termType xrr:RdfBag;` `  ];` `  ];`
Generated triples	`<http://example.org/Dell> ex:builds [` `  a rdf:Bag;` `  rdf:_1 "Laptop";` `  rdf:_2 "Desktop".` `] .`

Unlike RDF terms of type IRI or blank node, RDF terms of type RDF collection or container cannot be used as subject or predicate of an RDF triple, nor as a graph IRI. Consequently:

> ***Definition 31***. *A term map with term type xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt is an object map (hence it cannot be a subject map, predicate map nor graph map).*
>
> *Formally:*
>
> *?X an rr:TermMap.*
> *?X rr:termType ?tt.*
> *?tt is one of xrr:RdfList, xrr:RdfSeq, xrr:RdfBag or xrr:RdfAlt*
> *⇒ ?X an rr:ObjectMap.*

A nested term map (property `xrr:nestedTermMap`) can be used to specify a term type, language tag or data type of members of an RDF collection or container. The example below illustrates the usage of a nested term map to generate an RDF collection of IRIs (first example), and an RDF sequence of data-typed literals (second example):

Input data	`{ "key1": ["url1", "url2"] }`	`{ "key1": [10, 20] }`
Term map	`[] rr:objectMap [` `  xrr:reference "$.key1.*";` `  rr:termType xrr:RdfList;` `  xrr:nestedTermMap [` `  rr:termType rr:IRI;` `  ];` `];`	`[] rr:objectMap [` `  xrr:reference "$.key1.*";` `  rr:termType xrr:RdfSeq;` `  xrr:nestedTermMap [` `  rr:termType rr:Literal;` `  rr:datatype xsd:integer;` `  ];` `];`
Generated RDF terms	In Turtle abbreviated notation: `(<url1> <url2>)`	`[ a rdf:Seq;` `  rdf:_1 10^^xsd:integer;` `  rdf:_2 20^^xsd:integer.` `];`

In a template-valued term map, the xrr:nestedTermMap property applies to values resulting from the application of the template string to the input values. In the first example below, term type rr:IRI applies to the result of the template string. The same principle applies in the second example with term type rr:Literal and datatype xsd:string.

Input data	`{` `  "FirstNames": '["John", "Albert"]',` `  "LastName": "Smith"` `}`	`{` `  "FirstNames": '["John", “Albert"]',` `  "LastName": "Smith"` `}`
Term map	`[] rr:objectMap [` `  rr:template "http://example.org/` `  {$.FirstNames.*}/{$.LastName}";` `  rr:termType xrr:RdfList;` `  xrr:nestedTermMap [` `  rr:termType rr:IRI;` `  ];` `];`	`[] rr:objectMap [` `  rr:template` `  "{$.FirstNames.*} {$.LastName}";` `  rr:termType xrr:RdfList;` `  xrr:nestedTermMap [` `  rr:termType rr:Literal;` `  rr:datatype xsd:string;` `  ];` `];`
Generated RDF terms	`(<http://example.org/John/Smith>` `  <http://example.org/Albert/Smith>)`	`( "John Smith"^^xsd:string` `  "Albert Smith"^^xsd:string )`

B.3.3 Parsing nested structured values

The example below illustrates the use of a nested term map to (i) parse nested structured values ("teams" are collections of "team" elements, which are collections of "member" elements) and (ii) translate those nested structured values into RDF terms of class rdf:List.

Input data	<teams> <team> <member>John</member> <member>Paul</member> </team> <team> <member>Cathy</member> <member>Ed</member> </team> </teams>
Term map	[] rr:objectMap [xrr:reference "/teams/team"; xrr:nestedTermMap [xrr:reference "/member"; rr:termType xrr:RdfList;];]; The first xrr:reference property ("/teams/team") selects "team" elements from the XML input, each "team" element being the root of an XML tree whose descendants are "member" elements. The second xrr:reference property ("/member"), within the xrr:nestedTermMap property, is evaluated sequentially against the results of the parent reference expression. Thus, the xrr:RdfList term type successively applies to "member" elements of the first team, then to "member" elements of the second team. Finally the term map generates two RDF collections, one per team element.
Generated RDF terms	("John" "Paul") ("Cathy" "Ed") *Note*: the object map has no rr:termType property, therefore its term type is that of its nested term type, that is xrr:RdfList.

The subsequent example generates one RDF sequence of nested RDF collections. Elements of the inner RDF collections are typed as rr:Literal and assigned a language tag using a second nested xrr:nestedTermMap property.

Input data	{ "teams": [["John", "Paul"] , ["Cathy", "Ed"]] }
Term map	[] rr:objectMap [xrr:reference "$.teams.*"; rr:termType xrr:RdfSeq; # represent "teams" as an rdf:Seq # Describe the elements of the RDF sequence xrr:nestedTermMap [rr:termType xrr:RdfList; # represent each team as an rdf:List # Type members of each team as literals with language "en" # using a simple nested term map xrr:nestedTermMap [rr:template "Player {$.*}";

	rr:termType rr:Literal; rr:language "en";];];];
Generated RDF terms	[a rdf:Seq; rdf:_1 ("Player John"@en "Player Paul"@en); rdf:_2 ("Player Cathy"@en "Player Ed"@en);]

As already mentioned, in a template-valued term map, property `xrr:nestedTermMap` applies to values resulting from the application of the template string to input values. Thus, defining a nested term map in a template-valued term map suggests that the template produces a valid expression with regards to the current data format, that, in turn, is interpreted against a path expression provided by an xrr:reference or rr:template property.

For instance, applying the template string:

```
'\{ "first": "{FirstNames}", "last": "{LastName}" \}'
```

would produce a string formatted as a JSON dictionary, like:

```
{ "first": "John", "last": "Smith" }
```

This use case is illustrated in the example below:

Input data	{ "FirstNames": '["John", "Albert"]', "LastName": "Smith" }
Term map	[] rr:objectMap [rr:template '\{ "first": "{$.FirstNames.*}", "last": "{$.LastName}" \}'; xrr:nestedTermMap [xrr:reference "$.*"; rr:termType xrr:RdfList;];]
Generated RDF terms	("John" "Smith") ("Albert" "Smith") Two values are generated by applying the template string; they are formatted as JSON arrays: { "first": "John", "last": "Smith" } { "first": "Albert", "last": "Smith" } The `xrr:nestedTermMap` property instructs to parse those values using the JSONPath expression "$.*" (property `xrr:reference`), and generates an RDF collection (`rdf:List`) for each of them.

> *Note*: this use case may seem rather awkward and probably of little use, but insofar as it is consistent with the xR2RML language definition, we think it should be considered as valid.

B.3.4 Multiple Mapping Strategies

The flexibility offered by nested term maps allows the same mapping to be written using various strategies: path expressions of properties `xrr:reference` and rr:template can be split in several levels of term map and nested term map.

For instance, both term maps below produce equivalent results. In the first case (left), the JSONPath expression ($.teams.*.*) retrieves all team members at once. In the second case (right), teams are retrieved first ($.teams.*), then the `xrr:nestedTermMap` property runs a second JSONPath evaluation to retrieve and datatype team members.

<table>
<tr><td>Input data</td><td colspan="2">{ "teams": [["John", "Paul"] , ["Cathy", "Ed"]] }</td></tr>
<tr><td>Term maps</td><td>[] xrr:logicalSource […];
rr:objectMap [
xrr:reference "$.teams.*.*";
rr:datatype xsd:string;
];</td><td>[] xrr:logicalSource […];
rr:objectMap [
xrr:reference "$.teams.*";
xrr:nestedTermMap [
xrr:reference "$.*";
rr:datatype xsd:string;
];
];</td></tr>
<tr><td>Generated RDF terms</td><td colspan="2">"John"^^xsd:string
"Paul"^^xsd:string
"Cathy"^^xsd:string
"Ed"^^xsd:string</td></tr>
</table>

It is likely that the first case will be more efficient as only one JSONPath expression is evaluated, whereas in the second case two JSONPath expressions are evaluated in sequence.

Similarly, the example below shows how a mixed-syntax path can be split into a term map and a nested term map:

<table>
<tr><td>[] xrr:logicalSource […];
rr:objectMap [
xrr:reference
"Column(col)/XPath(\\/person\\/name)";
rr:datatype xsd:string;
];</td><td>[] xrr:logicalSource […];
rr:objectMap [
rr:column "col";
xrr:nestedTermMap [
xrr:reference
"XPath(\\/person\\/name)";
rr:datatype xsd:string;
];
];</td></tr>
</table>

Both mappings are likely to be equally efficient, as both evaluations (column selection and XPath expression evaluation) need to be done anyway.

B.3.5 Default Term Types

This section is an adaptation of section 7.4 (http://www.w3.org/TR/r2rml/#termtype) of the R2RML specification. xR2RML additions to R2RML are highlighted.

If the term map has an optional rr:termType property then its term type is the value of that property. The value MUST be one of the following options:

- If the term map is a subject map: rr:IRI or rr:BlankNode
- If the term map is a predicate map: rr:IRI
- If the term map is an object map: rr:IRI, rr:BlankNode, rr:Literal, rdf:List, rdf:Seq, rdf:Bag, rdf:Alt.
- If the term map is a graph map: rr:IRI.

If the term map does not have an rr:termType property, then its term type is:

- rr:Literal, if it is an object map and at least one of the following conditions is true:
  - It is a column-based term map.
  - It has an rr:language property (and thus a specified language tag).
  - It has an rr:datatype property (and thus a specified datatype).
  - It does not have an rr:language property and it has a nested term map that has an rr:language property.
  - It does not have an rr:datatype property and it has a nested term map that has an rr:datatype property.
  - the term type of the value of its nested term map.
- rr:IRI, otherwise.

A corollary of this definition is that the xrr:nestedTermMap property may be used in a subject map, predicate map or graph map only if it produces IRIs. Consequently:

> ***Definition 32***. *A term map with an `xrr:nestedTermMap` property may be a subject map or graph map only if (i) it does not have an rr:termType property and (ii) its nested term map has an `rr:termType` property with object `rr:IRI` or `rr:BlankNode`.*
>
> *A term map with an `xrr:nestedTermMap` property may be a predicate map only if (i) it does not have an `rr:termType` property and (ii) its nested term map property has an `rr:termType` property with object `rr:IRI`.*

B.4 Reference relationships between logical sources

The following definitions are an adaptation of R2RML specification section 8 (http://www.w3.org/TR/r2rml/#foreign-key). xR2RML additions to R2RML are highlighted.

> ***Definition 33***. *A referencing object map allows using the subjects of another triples map as the objects generated by a predicate-object map. Since both triples maps may be based on different logical sources, this may require a join between the logical sources.*
>
> *A referencing object map resource has exactly one `rr:parentTriplesMap` property (its value is a triples map), and optional rr:joinCondition properties. A join condition has exactly `one rr:child` property and one rr:parent property. The `rr:child` property references the join condition's child data element, the rr:parent property references the join condition's parent data element. Data*

element references are valid path expressions with regards to the reference formulation, possibly using mixed-syntax paths.

A referencing object map may have an rr:termType property with an RDF collection or container term type (see further details in §B.4.2).

The child query of a referencing object map is the query or source name of the logical source of the triples map containing the referencing object map.

The parent query of a referencing object map is the query or source name of the logical source of the referencing object map's parent triples map.

Properties `rr:child` and `rr:parent` use valid path expressions to reference data elements. As described in §B.3.2.3, such path expressions may produce multiple terms. Consequently, the equivalent join query of a referencing object map must take into account the fact that child and parent references are multi-valued. More precisely, a join between two multi-valued references should be satisfied if at least one data element of the first reference matches one data element of the second reference.

The join query of a referencing object map is defined below using SQL syntax (SELECT... FROM... AS... WHERE) and first order logic for the description of WHERE conditions:

***Definition 34**. If a referencing object map has no join condition, its join query is:*

*SELECT * FROM ({child-query}) AS tmp*

If an xR2RML referencing object map has at least one join condition, its equivalent join query is defined as follows:

*SELECT * FROM ({child-query}) AS **child**, ({parent-query}) AS **parent***
WHERE
$\exists c_1 \in eval(\textbf{child}, \{child\text{-}ref_1\}), \exists p_1 \in eval(\textbf{parent}, \{parent\text{-}ref_1\}), c_1 = p_1$
AND
$\exists c_2 \in eval(\textbf{child}, \{child\text{-}ref_2\}), \exists p_2 \in eval(\textbf{parent}, \{parent\text{-}ref_2\}), c_2 = p_2$
AND ...

where $\{child\text{-}ref_n\}$ and $\{parent\text{-}ref_n\}$ are the child and parent references of the n^{th} join condition, and eval(source, {ref}) is the result of evaluating the data element reference "{ref}" against data "source".

Note: when applied to a relational database, in which child and parent references are single-valued, this definition can be simplified into the R2RML join query definition:

```
SELECT * FROM ({child-query}) AS child, ({parent-query}) AS parent
WHERE child.{child-ref1} = parent.{parent-ref1}
AND   child.{child-ref2} = parent.{parent-ref2}
AND …
```

B.4.1 Reference relationship with structured values

The relational database example below models the relation between medical doctors and the studies for which they are investigators. Column "Doctor.studies" contains JSON arrays whereof elements are references to column "Study.study_id".

<table>
<tr><td>Input data</td><td>Table Study
<table>
<tr><td>study_id</td><td>study_name</td></tr>
<tr><td>1</td><td>study1</td></tr>
<tr><td>2</td><td>study2</td></tr>
<tr><td>3</td><td>study3</td></tr>
</table>

Table Doctor
<table>
<tr><td>doc_id</td><td>doc_name</td><td>studies</td></tr>
<tr><td>1</td><td>D1</td><td>[1,2]</td></tr>
<tr><td>2</td><td>D2</td><td>[3]</td></tr>
</table>
</td></tr>
<tr><td>Mapping graph</td><td>

```
<#Study>
    rr:logicalTable [ rr:tableName "Study" ];
    rr:subjectMap [rr:template "http://example.org/study/{study_name}" ].

<#Doctor>
    rr:logicalTable [ rr:tableName "Doctor" ];
    rr:subjectMap [
        rr:template "http://example.org/doc/{doc_name}";
    ];
    rr:predicateObjectMap [
        rr:predicate ex:investigatorOf;
        rr:objectMap [
            rr:parentTriplesMap <#Study>;
            rr:joinCondition [
                rr:parent "study_id";
                rr:child "Column(studies)/JSONPath($.*)";
            ];
        ];
    ].
```

The rr:child property uses a mixed-syntax path specifying that the data retrieved is formatted in JSON, and that each element of this structured value is considered in the join operation.</td></tr>
<tr><td>Generated triples</td><td>

```
<http://example.org/doc/D1> ex:investigatorOf
   <http://example.org/study/study1>, <http://example.org/study/study2> .
<http://example.org/doc/D2> ex:investigatorOf
   <http://example.org/study/study3> .
```

According to the equivalent join query definition, the join query is as follows ("child" and "parent" notations have been removed for readability):

SELECT * FROM Doctor, Study
WHERE ∃c ∈ eval(Doctor, Column(studies)/JSONPath($.*)),
∃p ∈ Study.study_id,
c = p</td></tr>
</table>

	where *eval(Doctor, Column(studies)/JSONPath($.*))* represents the evaluation of mixed-syntax path "Column(studies)/JSONPath($.*)" on table Doctor. Since Study.study_id is single-valued, we can rewrite the query as: SELECT * FROM Doctor, Study WHERE ∃ c ∈ Doctor.Column(studies)/JSONPath($.*), c = Studies.study_id The join query results in this table:

doc_id	doc_name	studies	study_id	study_name
1	D1	[1,2]	1	study1
1	D1	[1,2]	2	study2
2	D2	[3]	3	study3

B.4.2 Generating RDF collection/container with a referencing object map

In R2RML, referencing object maps do not have an rr:termType property as they should only produce RDF terms of type rr:IRI. In xR2RML however, the result of a join query may be translated into an RDF collection or container using property rr:termType. The rr:termType has a specific semantics here: it groups join query results by subjects of the generated triples, i.e. by child reference, and renders all objects in the same grouping as an RDF collection or container.

> ***Definition 35****. If a referencing object map has no* `rr:termType` *property, then its term type is rr:IRI (compliant with the R2RML definition).*
>
> *A referencing object map may have an* `rr:termType` *property with an RDF collection or container term type (*`xrr:RdfList`*,* `xrr:RdfSeq`*,* `xrr:RdfBag` *or* `xrr:RdfAlt`*). In that case, members of the collection or container are necessarily of type* `rr:IRI`*.*
>
> *In a referencing object map with an RDF collection or container term type, results of the join query pertaining to the same subject term are grouped together. The objects of the triples map are grouped in a single object of type RDF collection or container, as instructed by the* `rr:termType` *property.*

In the example below the referencing object map has an `rr:termType` property with value xrr:RdfList:

Input data	JSON documents retrieved by the query in the <#Study> triples map: `{ "study_id":1, "study_name":"study1" }` `{ "study_id":2, "study_name":"study2"}` `{ "study_id":3, "study_name":"study3"}` JSON documents retrieved by the query in the <#Doctor> triples map: `{ "doc_name":"D1", "studies": [1,2] }` `{ "doc_name":"D2", "studies": [2,3] }`

<table>
<tr><td>Mapping graph</td><td>Below, queries to retrieve Studies and Doctors are referred to as <Study query> and <Doctor query>.

<pre>
<#Doctor>
  xrr:logicalSource [xrr:query "<Doctor query>";];
  rr:subjectMap [rr:template "http://example.org/doc/{$.doc_name}"].

<#Study>
  xrr:logicalSource [xrr:query "<Study query>";];
  rr:subjectMap [rr:template http://example.org/study/{$.study_name}"];
  rr:predicateObjectMap [
    rr:predicate ex:hasInvestigators;
    rr:objectMap [
      rr:parentTriplesMap <#Doctor>;
      rr:joinCondition [
        rr:child "$.study_id";
        rr:parent "$.studies.*";
      ];
      rr:termType xrr:RdfList;
    ];
  ].
</pre>
</td></tr>
<tr><td>Generated RDF triples</td><td>
<pre>
<http://example.org/study/study1> ex:hasInvestigators
        (<http://example.org/doc/D1>).
<http://example.org/study/study2> ex:hasInvestigators
        (<http://example.org/doc/D1>
          <http://example.org/doc/D2>).
<http://example.org/study/study3> ex:hasInvestigators
        (<http://example.org/doc/D2>).
</pre>

Explanation: according to the equivalent join query definition, the join query is as follows:
SELECT * FROM (<Study query>) as child,
(<Doctor query>) as parent
WHERE ∃ p ∈ eval(parent, $.studies.*),
p = eval(child, $.study_id)

where eval(parent, $.studies.*)</i> represents the evaluation of path "$.studies.*" on the result of the parent query, and eval(child, $.study_id)</i> represents the evaluation of path "$.study_id" on the result of the child query.

The equivalent join query results in the following documents:
<pre>
{"study_id":1, "study_name":"study1", "doc_name":"D1", "studies": [1,2]}
{"study_id":2, "study_name":"study2", "doc_name":"D1", "studies": [1,2]}
{"study_id":2, "study_name":"study2", "doc_name":"D2", "studies": [2,3]}
{"study_id":3, "study_name":"study3", "doc_name":"D2", "studies": [2,3]}
</pre>
Then, term type <code>xrr:RdfList</code> instructs to group results pertaining to the same subject, i.e. the same "study_id".
</td></tr>
</table>

B.4.3 Generating RDF collection/container with a referencing object map in the relational case

An interesting consequence of using the `rr:termType` in a referencing object map is the ability, in the case of a relational database with standard SQL values, to build an RDF collection or container reflecting a one-to-many relation. In the example below, foreign key Study.doctor relates each study to its investigator in a many-to-one relation (several studies may have the same investigator). Considered the other way round, it can be seen as a one-to-many relation (one doctor investigates several studies). The mapping graph describes the generation of each doctor along with the list of studies he/she investigates.

Input data

Table Study

study_id	study_name	doctor
1	study1	1
2	study2	1
3	study3	2

Table Doctor

doc_id	doc_name
1	D1
2	D2

Mapping graph

```
<#Study>
    rr:logicalTable [ rr:tableName "Study" ];
    rr:subjectMap [
        rr:template "http://example.org/study/{study_name}";
    ].

<#Doctor>
    rr:logicalTable [ rr:tableName "Doctor" ];
    rr:subjectMap [
        rr:template "http://example.org/doc/{doc_name}";
    ].
    rr:predicateObjectMap [
        rr:predicate ex:investigatesStudies;
        rr:objectMap [
            rr:parentTriplesMap <#Study>;
            rr:joinCondition [
                rr:child "doc_id";
                rr:parent "doctor;
            ];
            rr:termType xrr:RdfList;
        ];
    ].
```

Generated RDF triples

```
<http://example.org/doc/D1> ex: investigatesStudies
   (<http://example.org/study/study1>
    <http://example.org/study/study2>) .
<http://example.org/doc/D2> ex: investigatesStudies
    ( <http://example.org/study/study3> ) .
```

The equivalent join query results in this table:

doc_id	doc_name	study_id	study_name	doctor
1	D1	1	study1	1
1	D1	2	study2	1
2	D2	3	study3	2

Results are grouped by subject, i.e. by column "doc_name" to generate RDF lists.

Appendix C. Translation of a Triple Pattern into Abstract Atomic Queries

In this appendix, we provide the detailed algorithm of functions used in the $transTP_m$ function, defined in section 6.6.

C.1 Functions genProjection and genProjectionParent

In Algorithm 9, we first describe function *getReferences*, a utility function used in subsequent functions.

Algorithm 9: Function getReferences returns the xr2RML data element references associated with an xR2RML term map

```
Function getReferences(termMap):
  case type(termMap)
    template-valued   : termVal ← getTemplateReferences(termMap.template)
    reference-valued : termVal ← termMap.reference
    constant-valued   : termVal ←  termMap.constant
  end case
  return termVal
```

Algorithm 10: Generate the list of xR2RML data element references that must be projected in the abstract query. A variable is project with the AS abstract language keyword: <xR2RML reference> AS ?variable

Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp.

Function *genProjection*(tp, TM):

```
refList ← <empty list>

if type(tp.sub) is VARIABLE then
   refList ← refList | getReferences(TM.subjectMap) AS tp.sub
end if

if type(tp.pred) is VARIABLE then
   refList ← refList | getReferences(TM.predicateObjectMap.predicateMap) AS tp.pred
end if

OM ← TM.predicateObjectMap.objectMap
if OM is a ReferencingObjectMap then
   // Since we do not know the target database,
   // the join may have to be computed by the query processing engine.
   // Hence, the joined fields are always projected, whether tp.obj is an IRI or a variable:
   refList ← refList | getReferences(OM.joinCondition.child)

else if type(tp.obj) is VARIABLE then
   refList ← refList | getReferences(OM) AS tp.obj
end if
return refList
```

Algorithm 11: Generates the list of xR2RML data element references from a parent triples map that must be projected in the abstract query

Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp, its object map is a referencing object map (it refers to a parent triples map).

Function *genProjectionParent*(tp, TM):

```
refList ← <empty list>
ROM ← TM.predicateObjectMap.objectMap    // Referencing Object Map

// Joined fields are always projected, whether tp.obj is an IRI or a variable:
refList ← refList | getReferences(ROM.joinCondition.parent)

// If tp.obj is a variable, the subject of the parent TM is projected too
if type(tp.obj) is VARIABLE then
   refList ← refList | getReferences(ROM.parentTriplesMap.subjectMap) AS tp.obj
end if
return refList
```

C.2 Functions genCond and genCondParent

We first describe function *getValue* that is used in subsequent functions.

Algorithm 12: Function getValue returns the value of an RDF term, depending on the xR2RML term map where it is applied.

This is simply a utility function that applies the inverse expression in case of a template-valued term map, and returns the RDF term as is otherwise.

```
Function getValue(rdfTerm, termMap):
  case type(termMap)
    template-valued   : termVal ← inverseExpression(rdfTerm, termMap.inverseExpression)
    reference-valued : termVal ← rdfTerm
    constant-valued   : termVal ← rdfTerm
  end case
  return termVal
```

Algorithm 13: Generate abstract query conditions by matching a triple pattern with a triples map

```
Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp, f is a SPARQL filter.
Function genCond(tp, TM, f):

  cond ← <empty list>

  // Subject part
  if type(TM.subject) is reference-valued or template-valued then
    case type(tp.sub)

      IRI:
        cond ← cond | equals(getValue(tp.sub, TM.subjectMap), getReferences(TM.subjectMap))

      VARIABLE:
        if f contains a condition mentioning tp.sub then
          cond ← cond | sparqlFilter(getReferences(TM.subjectMap), f)
        else
          cond ← cond | isNotNull(getReferences(TM.subjectMap))
        end if
    end case
  end if

  // Predicate part
  PM ← TM.predicateObjectMap.predicateMap
  if type(PM) is reference-valued or template-valued then
    case type(tp.pred)

      IRI:
        cond ← cond | equals(getValue(tp.pred, PM), getReferences(PM))
```

```
        VARIABLE :
          if f contains a condition mentioning tp.pred then
            cond ← cond | sparqlFilter(getReferences(PM), f)
          else
            cond ← cond | isNotNull(getReferences(PM))
          end if
      end case
    end if

    // Object part
    OM ← TM.predicateObjectMap.objectMap
    case type(tp.obj)

      LITERAL:
        if type(OM) is reference-valued or template-valued then
          cond ← cond | equals(getValue(tp.obj, OM), getReferences(OM))
        end if

      IRI:
        if OM is a ReferencingObjectMap then
            cond ← cond | isNotNull(OM.joinCondition.child)
        else if type(OM) is reference-valued or template-valued then
            cond ← cond | equals(getValue(tp.obj, OM), getReferences(OM))
        end if

      VARIABLE:
        if OM is a ReferencingObjectMap then
            cond ← cond | isNotNull(OM.joinCondition.child)
        else if type(OM) is reference-valued or template-valued then
            if f contains a condition mentioning tp.obj then
              cond ← cond | sparqlFilter(getReferences(OM), f)
            else
              cond ← cond | isNotNull(getReferences(OM))
            end if
        end if
    end case
```

Algorithm 14: Generate abstract query conditions by matching the object of a triple pattern with a referencing object map

Input: tp is a triple pattern, TM is an xR2RML triples map bound to tp and its object map is a referencing object map (it refers to a parent triples map), f is a SPARQL filter.

Function genCondParent(tp, TM, f):

```
cond ← <empty list>
OM ← TM.predicateObjectMap.objectMap

case type(tp.obj)
    IRI:
        // tp.obj is a constant IRI to be matched with the subject of the parent TM:
        // add an equality condition for each reference in the subject map of the parent TM
        if type(OM.parentTriplesMap.subjectMap) is reference-valued or template-valued then
            obj_value ← getValue(tp.obj, OM.parentTriplesMap.subjectMap)
            cond ← cond | equals(obj_value, getReferences(OM.parentTriplesMap.subjectMap))
        end if
        // And in any case add a non null condition to satisfy the join
        cond ← cond | isNotNull(OM.joinCondition.parent)

    VARIABLE:
        // tp.obj is a SPARQL variable to be matched with the subject of the parent TM
        if type(OM.parentTriplesMap.subjectMap) is reference-valued or template-valued then
            if f contains a condition mentioning tp.obj then
                cond ← cond | sparqlFilter(getReferences(OM.parentTriplesMap.subjectMap), f)
            else
                cond ← cond | isNotNull(getReferences(OM.parentTriplesMap.subjectMap))
            end if
        end if
        // And in any case add a non null condition to satisfy the join
        cond ← cond | isNotNull(OM.joinCondition.parent)
end case
```

Bibliography

[Akil et al., 2011] Akil H., Martone M. E. & Van Essen D. C. (2011). Challenges and Opportunities in Mining Neuroscience Data. *Science* 331(6018):708–712.

[Arenas et al., 2012] Arenas M., Bertails A., Prud'hommeaux E. & Sequeda J. (2012). A Direct Mapping of Relational Data to RDF. *W3C Recommendation* <http://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/>.

[Ashok, 2009] Ashok M. (2009). W3C RDB2RDF Incubator Group Report. <http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/>.

[Auer et al., 2009] Auer S., Dietzold S., Lehmann J., Hellmann S. & Aumueller D. (2009). Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th International conference on World Wide Web*, pp. 621–630. Madrid, Spain.

[Ball, 2014] Ball A. (2014). *How to License Research Data*. Digital Curation Centre (DCC) <http://www.dcc.ac.uk/resources/how-guides/license-research-data>.

[Barrasa et al., 2004] Barrasa J., Corcho Ó. & Gómez-Pérez A. (2004). R2O, an extensible and semantically based database-to-ontology mapping language. In *Proceedings of the 2nd Workshop on Semantic Web and Databases (SWDB)* vol. 3372. Toronto, Canada. Springer-Verlag.

[Beckett et al., 2014] Beckett D., Berners-Lee T., Prud'hommeaux E. & Carothers G. (2014). RDF 1.1 Turtle: Terse RDF Triple Language. *W3C Recommendation* <https://www.w3.org/TR/2014/REC-turtle-20140225/>.

[Bedini et al., 2011] Bedini I., Matheus C., Patel-Schneider P. F., Boran A. & Nguyen B. (2011). Transforming XML schema to OWL using patterns. In *Proceedings of the Fifth IEEE International Conference on Semantic Computing (ICSC)*, pp. 102–109. Palo Alto, CA, USA. IEEE.

[Berners-Lee, 2006] Berners-Lee T. (2006). Linked Data, in Design Issues of the WWW. <http://www.w3.org/DesignIssues/LinkedData.html>.

[Bikakis et al., 2013] Bikakis N., Tsinaraki C., Gioldasis N., Stavrakantonakis I. & Christodoulakis S. (2013). The XML and Semantic Web Worlds: Technologies, Interoperability and Integration: a Survey of the State of the Art. In *Semantic Hyper/Multimedia Adaptation*, pp. 319–360. Springer.

[Bikakis et al., 2015] Bikakis N., Tsinaraki C., Stavrakantonakis I., Gioldasis N. & Christodoulakis S. (2015). The SPARQL2XQuery interoperability framework: Utilizing Schema Mapping, Schema Transformation and Query Translation to Integrate XML and the Semantic Web. *World Wide Web* 18(2):403–490.

[Bischof et al., 2012] Bischof S., Decker S., Krennwallner T., Lopes N. & Polleres A. (2012). Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics* 1(3):147–185.

[Bizer, 2009] Bizer C. (2009). The Emerging Web of Linked Data. *IEEE intelligent systems* 24(5):87–92.

[Bizer & Cyganiak, 2006] Bizer C. & Cyganiak R. (2006). D2R server - Publishing Relational Databases on the Semantic Web. In *Proceeding of the 5th International Semantic Web Conference (ISWC)*. Athens, GA, USA.

[Bizer & Cyganiak, 2014] Bizer C. & Cyganiak R. (2014). RDF 1.1 TriG. *W3C Recommendation* <https://www.w3.org/TR/2014/REC-trig-20140225/>.

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

[Bizer & Seaborne, 2004] Bizer C. & Seaborne A. (2004). D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*. Arlington, VA, USA.

[Blackwell & Green, 2000] Blackwell A. & Green T. R. (2000). A Cognitive Dimensions Questionnaire Optimised for Users. In *Proceedings of 12th Workshop of the Psychology of Programming Interest Group*, pp. 137–154. Corigliano Calabro, Cosenza, Italy.

[Bohring & Auer, 2005] Bohring H. & Auer S. (2005). Mapping XML to OWL Ontologies. *Leipziger Informatik-Tage* 72:147–156.

[Bonicel, 2013] Bonicel M. (2013). Biblissima : observatoire du patrimoine écrit du Moyen Âge et de la Renaissance. *Bulletin des bibliothèques de France* (5):23–26.

[Botoeva et al., 2016a] Botoeva E., Calvanese D., Cogrel B., Rezk M. & Xiao G. (2016). OBDA beyond relational DBs: A study for MongoDB. In *Proceedings of the 29th Int. Workshop on Description Logics*. Cape Town, South Africa.

[Botoeva et al., 2016b] Botoeva E., Calvanese D., Cogrel B., Rezk M. & Xiao G. (2016). *A formal presentation of MongoDB (Extended version)*. <http://arxiv.org/abs/1603.09291> [Accessed May 24, 2016].

[Breitling, 2009] Breitling F. (2009). A standard transformation from XML to RDF via XSLT. *Astronomical Notes* 330:755.

[Brewer, 2000] Brewer E. A. (2000). Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, *PODC '00*. Portland, OR, USA. ACM.

[Broca & Coriat, 2015] Broca S. & Coriat B. (2015). Le logiciel libre et les communs. *Revue internationale de droit économique* 29(3):265–284.

[Calbimonte et al., 2012] Calbimonte J.-P., Jeung H., Corcho O. & Aberer K. (2012). Enabling Query Technologies for the Semantic Sensor Web. *International Journal on Semantic Web and Information Systems* 8(1):43–63.

[Callou et al., 2015] Callou C., Michel F., Faron-Zucker C., Martin C. & Montagnat J. (2015). Towards a Shared Reference Thesaurus for Studies on History of Zoology, Archaeozoology and Conservation Biology. In *Semantic Web For Scientific Heritage (SW4SH), Workshops of the Extended Semantic Web Conferene (ESWC)*. Portoroz, Slovenia.

[Cerbah, 2008] Cerbah F. (2008). Learning highly structured semantic repositories from relational databases: The RDBToOnto tool. In *Proceedings of the 5th Extended Semantic Web Conference (ESWC)*, pp. 777–781. Athens, GA, USA.

[Chang et al., 2008] Chang F., Dean J., Ghemawat S., Hsieh W. C., Wallach D. A., Burrows M., Chandra T., Fikes A. & Gruber R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26(2):4.

[Chebotko et al., 2009] Chebotko A., Lu S. & Fotouhi F. (2009). Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering* 68(10):973–1000.

[Connolly, 2007] Connolly D. (2007). Gleaning Resource Descriptions from Dialects of Languages (GRDDL). *W3C Recommendation* <https://www.w3.org/TR/2007/REC-grddl-20070911/>.

[Corby et al., 2012] Corby O., Gaignard A., Zucker C. F. & Montagnat J. (2012). KGRAM versatile inference and query engine for the web of linked data. In *Proceedings of the int. Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pp. 121–128. Macau, China. IEEE Computer Society.

[Corby & Zucker, 2010] Corby O. & Zucker C. F. (2010). The KGRAM Abstract Machine for Knowledge Graph Querying. In *Proceedings of the International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, pp. 338–341. Toronto, Canada. IEEE.

[Cullot et al., 2007] Cullot N., Ghawi R. & Yetongnon K. (2007). DB2OWL : A Tool for Automatic Database-to-Ontology Mapping. In *Proceedings of the 15th Italian Symposium on Advanced Database Systems (SEBD)*, pp. 491–494. Torre Canne, Fasano, BR, Italy.

[Cyganiak, 2013] Cyganiak R. (2013). TARQL Mapping Language. <https://github.com/tarql/tarql/wiki/TARQL-Mapping-Language>.

[Cyganiak et al., 2012] Cyganiak R., Bizer C., Maresch O. & Becker C. (2012). The D2RQ Mapping Language v0.8. <http://d2rq.org/d2rq-language>.

[Cyganiak et al., 2014] Cyganiak R., Wood D. & Lanthaler M. (2014). RDF 1.1 Concepts and Abstract Syntax. *W3C Recommendation* <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.

[Das et al., 2012] Das S., Sundara S. & Cyganiak R. (2012). R2RML: RDB to RDF Mapping Language. *W3C Recommendation* <http://www.w3.org/TR/2012/REC-r2rml-20120927/>.

[Dean & Ghemawat, 2010] Dean J. & Ghemawat S. (2010). MapReduce: A Flexible Data Processing Tool. *Commun. ACM* 53(1):72–77.

[Debruyne & O'Sullivan, 2016] Debruyne C. & O'Sullivan D. (2016). R2RML-F: Towards Sharing and Executing Domain Logic in R2RML Mappings. In *Proceedings of the 9th Workshop on Linked Data on the Web (LDOW)*. Montreal, Canada.

[Dell'Aglio et al., 2014] Dell'Aglio D., Polleres A., Lopes N. & Bischof S. (2014). Querying the web of data with XSPARQL 1.1. In *Proceedings of the Developers Workshop, International Semantic Web Conference (ISWC)*, pp. 113–118. Seattle, WA, USA.

[Dimou et al., 2013] Dimou A., Sande M. V., Colpaert P., Mannens E. & Van de Walle R. (2013). Extending R2RML to a source-independent mapping language for RDF. In *Posters & Demos,*

12th International Semantic Web Conference (ISWC) vol. 1035, pp. 237–240. Sydney, Australia.

[Dimou et al., 2014a] Dimou A., Sande M. V., Colpaert P., Verborgh R., Mannens E. & Van de Walle R. (2014). RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *Proceedings of the 7th Workshop on Linked Data on the Web (LDOW)*. Seoul, Korea.

[Dimou et al., 2014b] Dimou A., Sande M. V., Slepicka J., Szekely P., Mannens E., Knoblock C. & Walle R. V. de. (2014). Mapping Hierarchical Sources into RDF Using the RML Mapping Language. In *Proceedings of the International Conference on Semantic Computing (ICSC)*, pp. 151–158. IEEE.

[Doan et al., 2012] Doan A., Halevy A. & Ives Z. G. (2012). *Principles of data integration*. Morgan Kaufmann <http://store.elsevier.com/Principles-of-Data-Integration/AnHai-Doan/isbn-9780124160446/>.

[Elliott et al., 2009] Elliott B., Cheng E., Thomas-Ogbuji C. & Ozsoyoglu Z. M. (2009). A complete translation from SPARQL into efficient SQL. In *Proceedings of the International Database Engineering & Applications Symposium*, pp. 31–42. ACM.

[Ermilov et al., 2016] Ermilov I., Lehmann J., Martin M. & Auer S. (2016). LODStats: The Data Web Census Dataset. In *Proceedings of the 15th International Semantic Web Conference (ISWC)*, pp. 38–46. Kobe, Japan. Springer.

[European Commission, 2016] European Commission. (2016). Guidelines on FAIR Data Management in Horizon 2020 v3.0. <http://ec.europa.eu/research/participants/data/ref/h2020/grants_manual/hi/oa_pilot/h2020-hi-oa-data-mgt_en.pdf>.

[Euzenat & Shvaiko, 2013] Euzenat J. & Shvaiko P. (2013). *Ontology Matching*. 2nd ed. Heidelberg (DE). Springer-Verlag.

[Fennell, 2014] Fennell P. (2014). Schematron - More useful than you'd thought. In *Proceeding of the XML London Conference*, pp. 103–112. London. XML London.

[Ferguson et al., 2014] Ferguson A. R., Nielson J. L., Cragin M. H., Bandrowski A. E. & Martone M. E. (2014). Big data from small data: data-sharing in the 'long tail' of neuroscience. *Nature Neuroscience* 17(11):1442–1447.

[Field et al., 2013] Field L., Suhr S., Ison J., Wittenburg P., Los W., Broeder D., Hardisty A., Repo S. & Jenkinson A. (2013). *Realising the full potential of research data: common challenges in data management, sharing and integration across scientific disciplines*. <http://zenodo.org/record/7636#.U4NQpCghF2A>.

[Floratou et al., 2014] Floratou A., Minhas U. F. & Ozcan F. (2014). SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *Proceedings of the VLDB Endowment* 7(12).

[Friedman et al., 1999] Friedman M., Levy A. Y., Millstein T. D. & others. (1999). Navigational Plans for Data Integration. *AAAI/IAAI* 1999:67–73.

[G8 Science Ministers, 2013] G8 Science Ministers. (2013). G8 Science Ministers Statement on efforts towards the transparency, coherence and coordination of the scientific research. <https://www.gov.uk/government/news/g8-science-ministers-statement>.

[Gajendran, 2013] Gajendran S. K. (2013). *A Survey on NoSQL Databases (technical report)*. <http://masters.donntu.edu.ua/2013/fknt/babich/library/article10.pdf>.

[Gardner et al., 2008] Gardner D., Akil H., Ascoli G. A., Bowden D. M., Bug W., Donohue D. E., Goldberg D. H., Grafstein B., Grethe J. S., Gupta A., Halavi M., Kennedy D. N., Marenco L., Martone M. E., Miller P. L., Müller H.-M., Robert A., Shepherd G. M., Sternberg P. W., Essen D. C. & Williams R. W. (2008). The Neuroscience Information Framework: A Data and Knowledge Environment for Neuroscience. *Neuroinformatics* 6(3):149–160.

[Gargominy et al., 2015] Gargominy P., Tercerie S., Régnier C., Ramage T., Dupont P., Vandel E., Daszkiewicz P., Poncet L. & Schoelink C. (2015). TAXREF v9. 0, référentiel taxonomique pour la France: Méthodologie, mise en oeuvre et diffusion. <https://inpn.mnhn.fr/docs-web/docs/download/136181>.

[Gibaud et al., 2011] Gibaud B., Kassel G., Dojat M., Batrancourt B., Michel F., Gaignard A. & Montagnat J. (2011). NeuroLOG: sharing neuroimaging data using an ontology-based federated approach. In *Proceedings of the AMIA Annual Symposium* vol. 2011, p. 472. Washington DC, USA.

[Görlitz & Staab, 2011] Görlitz O. & Staab S. (2011). SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *Proceedings of the 2nd International Workshop on Consuming Linked Data (COLD)*. Bonn, Germany.

[Haas et al., 1997] Haas L., Kossmann D., Wimmers E. & Yang J. (1997). Optimizing Queries across Diverse Data Sources. In *23rd International Conference on Very Large Data Bases (VLDB 1997)*, pp. 276–285. San Francisco, CA, USA.

[Hardin, 1968] Hardin G. (1968). The Tragedy of the Commons. *Science*.

[Harris & Seaborne, 2008] Harris S. & Seaborne A. (2008). SPARQL 1.0 Query Language. *W3C Recommendation* <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.

[Harris & Seaborne, 2013] Harris S. & Seaborne A. (2013). SPARQL 1.1 Query Language. *W3C Recommendation* <http://www.w3.org/TR/2013/REC-sparql11-query-20130321/>.

[He et al., 2007] He B., Patel M., Zhang Z. & Chang K. C.-C. (2007). Accessing the Deep Web. *Communications of the ACM* 50(5):94–101.

[Heath & Bizer, 2011] Heath T. & Bizer C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. 1st ed. Morgan & Claypool <http://linkeddatabook.com/>.

[Hecht & Jablonski, 2011] Hecht R. & Jablonski S. (2011). NoSQL Evaluation: A Use Case Oriented Survey. In *Proceedings of the International Conference on Cloud and Service Computing (CSC)*, pp. 336–341. Washington, DC, USA. IEEE Computer Society.

[Hert et al., 2011] Hert M., Reif G. & Gall H. C. (2011). A Comparison of RDB-to-RDF Mapping Languages. In *Proceedings of the 7th International Conference on Semantic Systems (I-Semantics)*, pp. 25–32. Graz, Austria. ACM.

[Husson, 2014] Husson A. (2014). Une sémantique statique pour MongoDB. In *Actes des 25th Journées Francophones des Langages Applicatifs (JFLA)*, pp. 77–92.

[Kauppinen & Espindola, 2011] Kauppinen T. & Espindola G. M. de. (2011). Linked Open Science-Communicating, Sharing and Evaluating Data, Methods and Results for Executable Papers. In *Proceedings of the International Conference on Computational Science (ICCS)* vol. 4, pp. 726–731. Singapore. Elsevier Procedia Computer Science.

[Keator et al., 2008] Keator D. B., Grethe J. S., Marcus D., Ozyurt B., Gadde S., Murphy S., Pieper S., Greve D., Notestine R., Bockholt H. J. & Papadopoulos P. (2008). A National Human Neuroimaging Collaboratory Enabled by the Biomedical Informatics Research Network (BIRN). *IEEE Transactions on Information Technology in Biomedicine* 12(2):162–172.

[Knoblock et al., 2012] Knoblock C. A., Szekely P., Ambite J. L., Goel A., Gupta S., Lerman K., Muslea M., Taheriyan M. & Mallick P. (2012). Semi-automatically mapping structured sources into the semantic web. In *Proceedings of the Extended Semantic Web Conference (ESWC)*, pp. 375–390. Heraklion, Greece. Springer.

[Kolev et al., 2014] Kolev B., Valduriez P., Jimenez-Peris R., Martínez-Bazan N. & Pereira J. (2014). CloudMdsQL: Querying Heterogeneous Cloud Data Stores with a Common Language. In *Proceeding of BDA2014: Gestion de Données – Principes, Technologies et Applications*. Autrans, France.

[Langegger & Wöss, 2009] Langegger A. & Wöss W. (2009). XLWrap–querying and integrating arbitrary spreadsheets with SPARQL. In *Proceedings of the 8th International Semantic Web Conference (ISWC)*. Washington. Springer.

[Larivière et al., 2015] Larivière V., Haustein S. & Mongeon P. (2015). The Oligopoly of Academic Publishers in the Digital Era. *PLoS ONE* 10(6):1–15.

[Le Ngoc et al., 2016] Le Ngoc L., Tireau A., Venkatesan A., Neveu P. & Larmande P. (2016). Development of a knowledge system for Big Data: Case study to plant phenotyping data. In *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics (WIMS)*, pp. 1–9. ACM.

[Lefrançois & Zimmermann, 2016] Lefrançois M. & Zimmermann A. (2016). Flexible RDF generation from RDF and heterogeneous data sources with SPARQL-Generate. In *Proceedings of the 20th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*. Bologna, Italy.

[Lenzerini, 2002] Lenzerini M. (2002). Data integration: A theoretical perspective. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 233–246. Madison, Wisconsin (USA). ACM.

[Lopes et al., 2011] Lopes N., Bischof S., Decker S. & Polleres A. (2011). On the Semantics of Heterogeneous Querying of Relational, XML and RDF Data with XSPARQL. In *Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA 2011)*. Lisbon, Portugal.

[Maali et al., 2011] Maali F., Cyganiak R. & Peristeras V. (2011). Re-using Cool URIs: Entity Reconciliation Against LOD Hubs. In *Proceedings of the Workshop on Linked Data on the Web (LDOW)* vol. 813. Hyderabad, India.

[Macina et al., 2016] Macina A., Montagnat J. & Corby O. (2016). Optimising SPARQL query processing in distributed knowledge graphs. In *Actes de la Conférence Gestion de Données - Principes, Technologies et Applications (BDA)*. Poitiers, France.

[Martone et al., 2004] Martone M. E., Gupta A. & Ellisman M. H. (2004). e-Neuroscience: Challenges and Triumphs in Integrating Distributed Data from Molecules to Brains. *Nature Neuroscience* 7(5):467–472.

[Max Planck Gesellschaft, 2003] Max Planck Gesellschaft. (2003). Berlin Declaration on Open Access to Knowledge in the Sciences and Humanities. <https://openaccess.mpg.de/Berlin-Declaration>.

[de Medeiros et al., 2015] de Medeiros L. F., Priyatna F. & Corcho O. (2015). MIRROR: Automatic R2RML Mapping Generation from Relational Databases. In *Engineering the Web in the Big Data Era* vol. 9114, *LNCS*, pp. 326–343. Springer.

[Michel et al., 2015] Michel F., Djimenou L., Faron-Zucker C. & Montagnat J. (2015). Translation of Relational and Non-Relational Databases into RDF with xR2RML. In *Proceeding of the 11th international conference on Web Information Systems and Technologies (WebIST)*, pp. 443–454. Lisbon, Portugal.

[Michel et al., 2016a] Michel F., Faron-Zucker C. & Montagnat J. (2016). Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference. In *Revised Selected Papers of the 11th international conference on Web Information Systems and Technologies (WebIST)*, *LNBIP*, pp. 275–296. Springer.

[Michel et al., 2016b] Michel F., Faron-Zucker C. & Montagnat J. (2016). A Generic Mapping-Based Query Translation from SPARQL to Various Target Database Query Languages. In *Proceeding of the 12th International Conference on Web Information Systems and Technologies (WebIST)* vol. 2, pp. 147–158. Rome, Italy.

[Michel et al., 2016c] Michel F., Faron-Zucker C. & Montagnat J. (2016). A Mapping-based Method to Query MongoDB Documents with SPARQL. In *Proceedings of the 27th International Conference on Database and Expert Systems Applications (DEXA)* vol. 9828, *LNCS*, pp. 52–67. Porto, Portugal. Springer.

[Michel et al., 2010] Michel F., Gaignard A., Ahmad F., Barillot C., Batrancourt B., Dojat M., Gibaud B., Girard P., Godard D., Kassel G. & others. (2010). Grid-wide neuroimaging data federation in the context of the NeuroLOG project. In *Proceddings of the HealthGrid Conference of Studies in health technology and informatics* vol. 159, pp. 112–123.

[Michel et al., 2014] Michel F., Montagnat J. & Faron-Zucker C. (2014). *A survey of RDB to RDF translation approaches and tools*. <http://hal.archives-ouvertes.fr/hal-00903568>.

[Miles & Bechhofer, 2009] Miles A. & Bechhofer S. (2009). SKOS Simple Knowledge Organization System Namespace Document. *W3C Recommendation* <https://www.w3.org/2009/08/skos-reference/skos.html>.

[Montoya et al., 2013] Montoya G., Ibáñez L. D., Skaf-Molli H., Molli P. & Vidal M.-E. (2013). *SemLAV: Local-as-View Mediation for SPARQL queries*. <http://hal.univ-nantes.fr/hal-00841985/> [Accessed March 3, 2014].

[Mugnier et al., 2016] Mugnier M.-L., Rousset M.-C. & Ulliana F. (2016). Ontology-Mediated Queries for NOSQL Databases. In *Proceedings of the 30th Conference on Artificial Intelligence (AAAI)*. Phoenix, Arizona, USA.

[Murray-Rust, 2008] Murray-Rust P. (2008). Open data in science. *Serials Review* 34(1):52–64.

[Ong et al., 2014] Ong K. W., Papakonstantinou Y. & Vernoux R. (2014). The SQL++ Unifying Semi-structured Query Language, and an Expressiveness Benchmark of SQL-on-Hadoop, NoSQL and NewSQL Databases (submitted). *CoRR* abs/1405.3631.

[Pavlo et al., 2009] Pavlo A., Paulson E., Rasin A., Abadi D. J., DeWitt D. J., Madden S. & Stonebraker M. (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pp. 165–178. Rhode-Island, USA. ACM.

[Pérez et al., 2009] Pérez J., Arenas M. & Gutierrez C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34(3):1–45.

[Pinkel et al., 2014] Pinkel C., Binnig C., Haase P., Martin C., Sengupta K. & Trame J. (2014). How to Best Find a Partner? An Evaluation of Editing Approaches to Construct R2RML Mappings. In *The Semantic Web: Trends and Challenges*, pp. 675–690. Springer.

[Poggi et al., 2008] Poggi A., Lembo D., Calvanese D., De Giacomo G., Lenzerini M. & Rosati R. (2008). Linking Data to Ontologies. In *Journal on Data Semantics 10*, pp. 133–173. Springer.

[Pollock et al., 2015] Pollock R., Tennison J., Kellogg G. & Herman I. (2015). Metadata Vocabulary for Tabular Data. *W3C Recommendation* <https://www.w3.org/TR/2015/REC-tabular-metadata-20151217/>.

[Pritchett, 2008] Pritchett D. (2008). BASE: An ACID Alternative. *Queue* 6(3):48–55.

[Priyatna et al., 2013] Priyatna F., Aranda C. B. & Corcho O. (2013). Applying SPARQL-DQP for federated SPARQL querying over Google Fusion Tables. In *Satellite Events of the Extended Semantic Web Conference (ESWC)*. Montpellier, France. Springer.

[Priyatna et al., 2014] Priyatna F., Corcho O. & Sequeda J. (2014). Formalisation and Experiences of R2RML-based SPARQL to SQL query translation using Morph. In *Proceeding of the World Wide Web Conference (WWW)*. Seoul, Korea.

[Rifkin, 2014] Rifkin J. (2014). *The Zero Marginal Cost Society*. St. Martin's Press.

[Rodríguez-Muro et al., 2013] Rodríguez-Muro M., Kontchakov R. & Zakharyaschev M. (2013). Ontology-based Data Access: Ontop of Databases. In *The Semantic Web–ISWC 2013*, pp. 558–573. Springer.

[Rodríguez-Muro & Rezk, 2015] Rodríguez-Muro M. & Rezk M. (2015). Efficient SPARQL-to-SQL with R2RML mappings. *Web Semantics: Science, Services and Agents on the World Wide Web* 33:141–169.

[Sahoo et al., 2009] Sahoo S., Halb W., Hellman S., Idehen K., Thibodeau T., Auer S., Sequeda J. & Ezzat A. (2009). A Survey of Current Approaches for Mapping of Relational Databases to RDF. <http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport_01082009.pdf>.

[Scharffe et al., 2012] Scharffe F., Atemezing G., Troncy R., Gandon F., Villata S., Bucher B., Hamdi F., Bihanic L., Képéklian G., Cotton F. & others. (2012). Enabling Linked Data Publication with the Datalift Platform. In *Proceedings of the AAAI Workshop on Semantic Cities*.

[Scholz, 2016] Scholz T. (2016). *Platform Cooperativism: Challenging the Corporate Sharing Economy*. Rosa Luxembourg Stiftung <http://vr-slide-uploads-live.s3.amazonaws.com/slides-294120160210-17746-14oy655.pdf>.

[Schwarte et al., 2011] Schwarte A., Haase P., Hose K., Schenkel R. & Schmidt M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In *10th International Conference on Semantic Web (ISWC'11)*, pp. 601–616. Springer.

[Schweik & Kitsing, 2010] Schweik C. M. & Kitsing M. (2010). Applying Elinor Ostrom's Rule Classification Framework to the Analysis of Open Source Software Commons. *Transnational Corporations Review* 2(1):13–26.

[Sengupta et al., 2013] Sengupta K., Haase P., Schmidt M. & Hitzler P. (2013). Editing R2RML Mappings Made Easy. In *Posters and demos of the 12th International Semantic Web Conference (ISWC)*. Sydney, Australia.

[Sequeda et al., 2014] Sequeda J. F., Arenas M. & Miranker D. P. (2014). OBDA: Query Rewriting or Materialization? In Practice, Both! In *The Semantic Web–ISWC 2014*, pp. 535–551. Springer.

[Sequeda & Miranker, 2013] Sequeda J. F. & Miranker D. P. (2013). Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web* 22:19–39.

[Sequeda et al., 2011] Sequeda J., Tirmizi S. H., Corcho Ó. & Miranker D. P. (2011). Survey of directly mapping SQL databases to the Semantic Web. *Knowledge Eng. Review* 26(4):445–486.

[Shvaiko & Euzenat, 2005] Shvaiko P. & Euzenat J. (2005). A survey of schema-based matching approaches. In *Journal on Data Semantics IV* vol. 3730, *LNCS*, pp. 146–171.

[Spanos et al., 2012] Spanos D.-E., Stavrou P. & Mitrou N. (2012). Bringing Relational Databases into the Semantic Web: A survey. *Semantic Web Journal* 3(2):169–209.

[Speicher et al., 2015] Speicher S., Arwe J. & Ashok M. (2015). Linked Data Platform 1.0. *W3C Recommendation* <https://www.w3.org/TR/2015/REC-ldp-20150226/>.

[Sporny et al., 2014] Sporny M., Longly D., Kellog G., Lanthaler M. & Lindström N. (2014). JSON-LD 1.0. A JSON-based Serialization for Linked Data. *W3C Recommendation* <https://www.w3.org/TR/2014/REC-json-ld-20140116/>.

[Stadler et al., 2015] Stadler C., Unbehauen J., Westphal P., Sherif M. & Lehmann J. (2015). Simplified RDB2RDF Mapping. In *Proceedings of the 8th Workshop on Linked Data on the Web (LDOW)*. Firenze, Italy.

[Stonebraker, 2012] Stonebraker M. (2012). New Opportunities for New SQL. *Communications of the ACM* 55(11):10–11.

[Stonebraker et al., 2010] Stonebraker M., Abadi D., DeWitt D. J., Madden S., Paulson E., Pavlo A. & Rasin A. (2010). MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM* 53(1):64–71.

[Tandy & Herman, 2015] Tandy J. & Herman I. (2015). Generating JSON from Tabular Data on the Web. *W3C Recommendation* <https://www.w3.org/TR/2015/REC-csv2json-20151217/>.

[Tandy et al., 2015] Tandy J., Herman I. & Kellogg G. (2015). Generating RDF from Tabular Data on the Web. *W3C Recommendation* <https://www.w3.org/TR/2015/REC-csv2rdf-20151217/>.

[Tercerie, 2016] Tercerie S. (2016). Web-services / TAXREF-MATCH ou comment réconcilier des jeux de données avec TAXREF. <http://seminaire-taxref.mnhn.fr/>.

[Tomaszuk, 2010] Document-oriented triplestore based on RDF/JSON. (2010). In *Logic, philosophy and computer science*, *Series: Studies in logic, grammar and rhetoric*, pp. 125–140. University of Białystok.

[Tounsi et al., 2015] Tounsi M., Zucker C. F., Zucker A., Villata S. & Cabrio E. (2015). Studying the history of pre-modern zoology with linked data and vocabularies. In *ESWC 2015, workshop Semantic Web For Scientific Heritage (SW4SH)*. Portoroz, Slovenia.

[Tournaire et al., 2011] Tournaire R., Petit J.-M., Rousset M.-C. & Termier A. (2011). Discovery of Probabilistic Mappings Between Taxonomies: Principles and Experiments. In *Journal on Data Semantics XV* vol. 6720, *LNCS*, pp. 66–101. Springer.

[UK Gov. Cabinet Office, 2013] UK Gov. Cabinet Office. (2013). G8 Open Data Charter. <https://www.gov.uk/government/publications/open-data-charter>.

[Unbehauen et al., 2013a] Unbehauen J., Stadler C. & Auer S. (2013). Accessing relational data on the web with SparqlMap. In *Semantic Technology*, pp. 65–80. Springer.

[Unbehauen et al., 2013b] Unbehauen J., Stadler C. & Auer S. (2013). Optimizing SPARQL-to-SQL Rewriting. In *Proceedings of Information Integration and Web-based Applications & Services (iiWAS)*, p. 324. Vienna, Austria. ACM.

[Verborgh et al., 2016] Verborgh R., Vander Sande M., Hartig O., Van Herwegen J., De Vocht L., De Meester B., Haesendonck G. & Colpaert P. (2016). Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web. *Journal of Web Semantics* 37–38:184–206.