Integrating Heterogeneous Data by Extending Semantic Web Standards

Nuno Lopes

Supervisor: **Priv.-Doz. Dr. Axel Polleres**

Internal Examiner: **Dr. Siegfried Handschuh**

External Examiner: **Prof. Dr. Claudio Gutiérrez**

Dissertation submitted in pursuance of the degree of Doctor of Philosophy

Digital Enterprise Research Institute, Galway
National University of Ireland, Galway / Ollscoil na hÉireann, Gaillimh

26th November 2012

The research presented herein was supported by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Acknowledgements

First of all I would like to thank Axel and Stefan for all their help, support, and directions provided during my Ph.D. Thanks to my examiners Claudio Gutiérrez and Siegfried Handschuh for their time.

Also a big thanks to all of my co-authors and co-workers for countless hours spent tackling interesting problems. Notably the persons I was closely working with: Stefan Bischof, Gergely Lukácsy, Antoine Zimmermann, Sabrina Kirrane, Umberto Straccia, Thomas Krennwallner, and other URQ members.

I would like to thank all the people that helped in reviewing and proofreading this thesis: Sabrina Kirrane, Deirdre Lee, Antoine Zimmermann, Alexandre Passant, Stefan Bischof, and Aidan Hogan. Especially thanks to Aidan and Sabrina for all their invaluable comments and their patience for my incessant nagging. Thanks also to Jürgen: it was great to have someone close to share the agony of thesis writing.

For the times when we needed some distraction from hard work the Foosball (thanks to the UDI2 members that supplied the table) or Magic The Gathering crowd was always close by!

Above all, I would like to thank my family and give credit to my partner Ana, for all the support and always understanding the (many) times when I had to prioritise work.

"To truly know something, you must become it. The trick is to not lose yourself in the process."

—**"Thirst for Knowledge", Magic the Gathering card**

Abstract

In enterprises different software applications are used to manage specific functions: customer relations, human resources, and manufacturing, each requiring specialised software. Relational databases are commonly used as the underlying storage mechanism for most of these software applications, often causing the same entities to be replicated in independent databases. In order to obtain an accurate overview of an enterprise, these independent data sources need to be combined. This hard task is commonly known as data integration and becomes even more difficult if we consider that the original data sources can be stored according to heterogeneous models. The Extensible Markup Language (XML) has become widely used on the World Wide Web (WWW) and in order to reuse Web data, XML needs to be included into the data integration process along side relational databases.

The Linking Open Data (LOD) initiative has also increased focus on another data model: the Resource Description Format (RDF). With the increasing availability of structured information on the Web, exposed following the Linked Data principles, RDF has also become an attractive format for representing integrated data, allowing existing enterprise data to be enriched, by connecting it to other data on the WWW.

Established approaches for data integration involve the development of custom applications that bridge the different sources and data formats. In this thesis we propose to make this bridge via a query and transformation language and propose optimisations for such a language that aim at reducing the execution times of the transformation queries.

RDF is already regarded as a useful format for representing integrated data but we argue that an extension of the RDF data model is necessary. This extension, which we call Annotated RDFS, allows us to represent domain-specific meta-information about the integrated data. For instance, defined Annotated RDFS domains allow temporal or provenance information to be maintained. Temporal information can help to determine the most up-to-date data, while provenance information can help to track information back to their original sources.

The language introduced in this thesis, called XSPARQL, combines different standard query languages – SQL, XQuery, and SPARQL – for accessing the heterogeneous data sources – relational, XML, and RDF data, respectively – and transforming between the different formats. The XSPARQL language also extends the SPARQL query language to allow for easily writing RDF transformations that can otherwise be cumbersome to write in SPARQL.

By further extending XSPARQL to support querying and creating Annotated RDFS, XSPARQL also allows meta-information to be extracted and attached to RDF triples. We illustrate this approach by introducing a use case where enterprise data from different systems is integrated and annotated with data from a novel Annotated RDFS domain: access control. This new domain maintains information regarding which agents are allowed to access the integrated information by replicating any access control information present in the original sources. We also propose a framework based on this new annotation domain that can enforce the access restrictions attached to each triple.

Declaration

I declare that this thesis is composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Nuno Lopes

30th July 2013

Contents

1. Introduction

The term *database* is commonly used to denote a *large collection of data* stored within a computer. While initial database systems focused mainly on the physical representation of the database, relying on files stored in the computers' filesystem, new database models were introduced that provided an abstraction layer over the physical representation of the database (Abiteboul, Hull et al., 1995; Silberschatz et al., 2005). One of these new database models, the relational model, is nowadays an almost-ubiquitous representation model and, since the introduction of this model by Codd (1970), there have been continuous advancements on storage and querying mechanisms for relational data. Several companies have focused on the commercialisation of relational database products, like Oracle, IBM DB2, and Microsoft SQL Server or open-source solutions like PostgreSQL and MySQL. The relational model relies on a strict separation between the data and the organisational schema of the data, where the schema must be provided beforehand to the *database management system*. An historical evolution of the data models in databases was presented by Navathe (1992) and, with the ultimate focus on graph databases, by Angles and Gutiérrez (2008a).

Database research also began to focus on different aspects of their data, for instance maintaining extra information such as temporal and provenance. Temporal information allows to determine when tuples were inserted into the database or to represent time periods when the tuple is considered valid. Provenance information becomes especially important when combining data from different sources, as it can be used determine from which sources information was derived from.

A timeline of the different data models, approaches for representing temporal and provenance information, and query languages is presented in Figure 1.1.[1] The aim of this figure is to show trends in research rather than exact dates for several topics. For example, research in semantic data models (like the Entity-Relationship model), temporality and provenance in databases, as well as graph databases, has spanned over several years.

Web Data Models

With the increased importance of the World Wide Web (WWW) in our daily lives, we have also witnessed a shift in the focus of enterprise applications: from the desktop to the Web (Abiteboul, Buneman et al., 1999). While the Web was initially used to boost the visibility of the enterprise, e.g. the corporate website quickly became an important form of attracting new business and clients, nowadays more enterprise tasks are accomplished though Web applications. For example, it is becoming commonplace for enterprises to use online calendars and meeting scheduling systems or even word processing systems that allow employees to collaboratively and concurrently work on the same document. Many of these Web applications follow a *multitier* approach, where data sources (often residing in relational databases) are integrated before exposing the result as HyperText Markup Language (HTML) pages, possibly linked to other information sources across the Web (Silberschatz et al., 2005; Abiteboul, Buneman et al., 1999).

For an open environment, such as the WWW, the predefined schema requirement of the relational model does not provide the flexibility necessary to deal with different representations of the same concepts and agreeing on a common representation of concepts (also referred to as a *global-schema*) is often not an achievable objective (Abiteboul, Buneman et al., 1999). Thus, *semi-structured* data emerged as a

[1] The models and languages we are particularly interested in for this thesis are highlighted in **bold**.

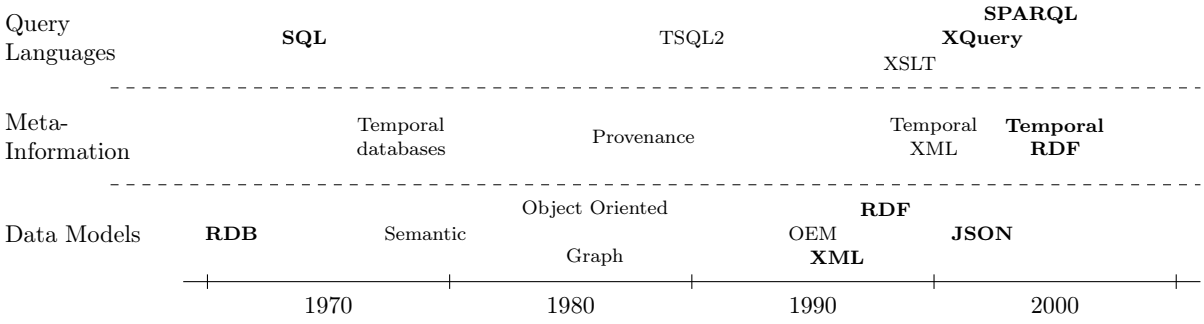


Figure 1.1.: Overview of data models and query languages

possible solution for avoiding the need for a predefined schema and several flexible data models, well suited for representing integrated data, were introduced (Papakonstantinou et al., 1995; Cluet et al., 1998; Buneman, 1997). Most of the presented data models for semi-structured data are tree or graph-based. On the WWW the Extensible Markup Language (XML) has become a widely used data representation format and is regarded by Abiteboul, Buneman et al. (1999) as the *de-facto* standard for information exchange.[2] XML follows a semi-structured, *tree-like* data model and several information integration projects relied on XML for representing their data (Draper et al., 2001b; Draper et al., 2001a; Baru et al., 1999; Manolescu et al., 2001; Yu and Popa, 2004).

Another data model, the Resource Description Framework (RDF)[3] (Manola and Miller, 2004) has recently been gaining importance on the Web and the Semantic Web (Berners-Lee, Hendler et al., 2001), supported by efforts like the Linking Open Data (LOD) initiative (Bizer, Heath et al., 2009). With the increase of data published in RDF as Linked Data, for example the DBpedia project (Bizer, Lehmann et al., 2009), a valuable and steadily growing source of structured information from various domains is being made available. The possibility of using LOD structured information in integration scenarios, allowing for an easy and low cost enrichment of enterprise data, provides further incentive for an enterprise to represent its integrated data in RDF (Stephens, 2007).

Data Segmentation and Data Integration

Four decades past the introduction of the relational model, most current software applications still rely on relational databases (RDB) to store their information. Enterprises commonly use RDB-based software applications to manage each aspect of their business, ranging from human resources to manufacturing. However, the use of specialised applications results in data segmentation, where the enterprise's valuable data is spread across different applications and relational databases (Dillnut, 2006; Silberschatz et al., 2005; P. A. Bernstein and Haas, 2008).

For instance, having an integrated view on customers allows an enterprise salesman to better target its product, or enables decision support systems to provide the management with a high-level view of all of the enterprise resources, from manufacturing to human resources and sales. As such, data integration in the context of relational databases has been a research topic in the past (Halevy, Rajaraman et al., 2006) and a good overview of integration techniques is provided by Doan and Halevy (2005). Focusing on enterprise data integration, common issues and possible solutions are described in Ziegler and Dittrich (2004); Halevy, Ashish et al. (2005).

Common approaches for integrating segmented data over relational database systems involve the use of

[2] More recently, JavaScript Object Notation (JSON) is becoming the preferred information exchange format, often in determent of XML.

[3] The RDF data model is considered essentially a *directed labelled graph*, however, in Section 1.2 we discuss different views on this representation model.

mediator or *data-warehousing* systems (Wiederhold, 1992; Abiteboul, Buneman et al., 1999; Ziegler and Dittrich, 2004). Mediator systems provide an abstraction layer over the original data sources often using a *global-schema* (or *mediated-schema*), where queries over this schema are executed over the original sources. On the other hand in the data-warehousing approach, data in the original sources is materialised into a common data model. Although both of these approaches have advantages and disadvantages, data-warehousing is particularly unsuitable in changing environments: consider for instance that one of the sources is highly dynamic e.g. containing data gathered from sensors: this would cause the integrated data to become quickly outdated (Abiteboul, Buneman et al., 1999). Other forms of data integration may also be considered, such as federated databases (Sheth and Larson, 1990), using Web services (Abiteboul, Benjelloun et al., 2002), or peer-to-peer systems (Arenas, Kantere et al., 2003).

For the scope of this thesis we consider performing the data integration by relying on a newly defined query language that is capable of accessing and transforming data stored in heterogeneous data sources and models that can be used as an implementation language for both mediator and data-warehousing scenarios.

Meta-information in the Data Integration Process

Although existing systems provide a way to solve the data segmentation problem, additional questions often arise when integrating data, such as *which sources* were involved in producing a specific piece of information or how to deal with *conflicting* information contained in the original sources. For example, different enterprise systems can store different addresses for an employee. There are several forms of dealing with conflicting information, for example, maintaing *provenance* information (also known as *lineage*) allows to determine from which of the original sources the specific information has been derived (Cui et al., 2000; Woodruff and Stonebraker, 1997; Benjelloun et al., 2008) in order to trace the origin of the contradiction and possibly correct it. Other approaches include maintaining *temporal* or *uncertain* information, which caters for evolving data, possibly avoiding contractions, and levels of confidence or certainty to be assigned to the conflicting data, respectively.[4] These aspects of data have been identified as an important part of the data integration process by Halevy, Rajaraman et al. (2006), and for example, the Trio system (Widom, 2005; Agrawal et al., 2006) extends the relational data model to consider both provenance and uncertain information.

Meta-information can become an important aspect of any data integration process and as such any suitable data model for representing integrated data needs to cater for this kind of information. Even aside from our core focus on data integration, meta-information is still an important aspect in any software application. It is common in applications and database schemas to maintain temporal information, for example, keeping logs of specific changes to the database, records of past employees, or having historical data available for manufacturing materials in order to predict future needs. In certain scenarios, temporal information even constitutes a critical aspect, where well known examples involve real-time monitoring such as air-traffic control. In these cases, temporal information is considered an important dimension, warranting its introduction into the relational model (Abiteboul, Hull et al., 1995; Snodgrass, 1999), which in turn lead to the concept of temporal databases.[5] Similar extensions have also been proposed to represent temporal information in XML (Amagasa et al., 2000; Rizzolo and Vaisman, 2008) and RDF (Gutiérrez, Hurtado and Vaisman, 2007; Pugliese et al., 2008; Tappolet and A. Bernstein, 2009). However temporal information is not the only kind of meta-information we can consider. Other extensions to the relational model also allow to represent ambiguous or approximate data in the form of fuzzy information. An overview of fuzzy databases is provided by Ma and Yan (2008), where fuzzy extensions were later also proposed for the XML (Ma and Yan, 2007) and RDF models (Straccia, 2009; Mazzieri

[4] In the following we refer to "information about data" commonly as *meta-information*.
[5] An historical overview of temporality in databases is presented by Snodgrass (1990).

and Dragoni, 2008; Lv et al., 2008).

1.1. Problem Statement

Currently established data models do not easily support the data integration process: a data model suitable for representing integrated data needs to be flexible and to cater for meta-information. However, even such a flexible data model is not enough for a complete data integration application: while a flexible data model facilitates the representation of integrated data, it still does not help the task of data gathering and transformation. For a complete solution, the data integration application must be aware of both the input sources and the target data model.

Existing solutions for enterprise data integration rely on specialised or custom-built applications, following either the mediator or data-warehouse approaches, to bridge the distributed sources and different data models. However, the costs of such applications quickly becomes too high (Halevy, Ashish et al., 2005). Another option is to consider using a query language for the data integration task (Draper et al., 2001a), but traditional query languages focus only on one data format and are thus not a possibility when the distributed data adheres to different data formats. In such cases, the use of a query language requires translating the original data into a common data model, much like the data-warehousing approach, and then performing the queries over the integrated data. A scenario that may also complicate such an approach is when the original data is protected by some form of access control, where these access restrictions would also need to be replicated in the mediator or data-warehouse in order to avoid information leakage.

With the introduction of the WWW, the data integration task can become unfeasible using traditional approaches due to the large amount of sources and different models. The evolution of the Web into the Semantic Web has also introduced a new data model, RDF, which can facilitate the representation of integrated data. However, the currently standardised RDF-based specifications do not cater for any type of meta-information regarding the individual RDF triples.[6]

1.2. A Model for Integrated Data

Software applications are now focused on the Web, having evolved from single-user applications and the personal computer. When we look at data models we find a similar evolution (cf. Figure 1.1), starting from data models that were mostly aimed at storing information in a single computer system to current ones that allow to share and link information to and from different sources. The widely disseminated relational data model, although perfect for a closed environment such as a specific application within an enterprise system, is not so well suited for open environments like the Web, or for representing integrated data since, in both cases, the data schema cannot always be determined *a priori*.

Given the variety of data and formats to be integrated, in this thesis we argue for a unifying data model and a query language capable of integrating data represented in different formats and models. Several semi-structured data models were presented that cater for dynamic, open, and flexible environments.

OEM. One of the most notable semi-structured data models is the Object Exchange Model (OEM) model (Papakonstantinou et al., 1995), defined in the context of the TSIMMIS data-integration project (Garcia-Molina et al., 1997). OEM is considered a semi-structured data model, consisting of a *graph of objects*. An object is represented as a quadruple $(label, oid, type, value)$, where *label* aims to be a human readable description of the object, thus making the data model *self-describing*. The *oid* is a unique

[6] One possible way of attaching meta-information to RDF triples is by using *reification*. However the use of this feature may be discouraged in future revisions of the RDF language cf. `http://www.w3.org/2010/09/rdf-wg-charter`.

identifier for each object and *type* indicates the type of the value. Finally, *value* consists of an atomic value or a set of objects.

XML. As presented by Suciu (1998), XML has important differences to semi-structured data, one of which is that XML more naturally represents data as *trees* whereas semi-structured data models, namely the OEM model, are usually graph-based.[7] Another core difference between XML and the OEM model resides in the *ordering* of the data model: XML is an intrinsically ordered data model, where each element has a specific order among its siblings, while the OEM model consists of an unordered graph similar (in terms of lack of ordering) to the relational model. The tree and ordered data model of XML makes integrating different documents a difficult task, even requiring Turing-complete languages for arbitrary transformations (Kepser, 2004). There are diverging opinions regarding whether XML is self-describing. Undoubtably, its unrestricted modelling features allow to specify the meaning of the data it contains, however, without the use of an XML Schema, it is impossible to accurately determine the types of the values (Siméon and Wadler, 2003). Finally, another difference between these models is the use of XML attributes and although the OEM model could be trivially extended to cater for similar representations, it does not have a natural equivalent (Suciu, 1998).

RDF. The RDF data model is closely related to the OEM data model for semi-structured data: (i) its representation model is a graph; (ii) it is unordered; and (iii) it is schema-less and self-describing, relying on Uniform Resource Identifiers (URIs) (Berners-Lee, Fielding et al., 2005) as unique identifiers for resources. The major differences between these data models is that the RDF data model is more correctly represented by an *hypergraph* since, as described by J. Hayes and Gutiérrez (2004), properties can themselves be the subject and object of other RDF triples, making the RDF model go beyond the theoretical notion of a graph. The existence of blank nodes in RDF, akin to existential variables, is another significant difference between the RDF and OEM data models. When compared to XML, RDF is schema-unware: the task of RDF Schema (RDFS) is to deduce implicit information rather than restricting the structure of the RDF data, as is the task of XML Schema for XML data. A survey of other graph models, focusing mostly on databases, is presented by Angles and Gutiérrez (2008a).

Requirements of a Data Model for Integrated Data

In the following we define the characteristics of a data model that is suitable for representing integrated data. Most importantly, any such data model needs to be *composable* i.e. the merging of data should be an easy task and, inspired by semi-structured data models, we can present the desired features of a data model for achieving composability as:

Entity-Centric Global Identifiers: The need for global identifiers is justified by the fact that we can, in any closed system, uniquely identify an entity, for example, by giving it a unique sequential identifier. For a global system, we also need to uniquely identify an entity, thus requiring a global identifier. In the case of RDF, this global identifier is the URI. For global identifiers, we need to make some assumptions, namely that they are used consistently i.e. the same identifier is not allowed to be used to identify different entities. It is however possible for the same entity to be identified by distinct identifiers.

Schema-less: A schema-less data model is one of the premises of semi-structured data: in an open and possibly global environment, obtaining agreement on the schema to represent any domain is a difficult and often impossible task. Monetary considerations aside (it would be extremely expensive

[7] XML does cater for representing arbitrary graphs, by assigning unique identifiers to nodes and then using XML *references* to point to those identifiers.

to develop a global schema) cultural differences or simply personal preferences often stand in the way of achieving agreement on a schema (Goh et al., 1994). Some common examples are the cultural, and often legislative, differences in the concepts of marriage: in some cultures marriage must be monogamous while for others this is not a requirement. Another example of schema conflicts would be modelling monotheistic and polytheistic views of religion. Conversely, under a closed environment, it is possible to obtain a level of agreement over a topic, for example all users of a specific system can agree on the use of a single *domain model*. For the WWW, one noteworthy attempt at defining a collection of models is `schema.org`; supported by the Google, Yahoo! and Bing search engines, the major incentive for using this vocabulary is that webpages will be better indexed by these search engines. However, the concepts defined so far are unambiguous, such as Places, Events, or Organisations. No vocabulary is yet provided that caters for ambiguous concepts such as those presented above.

Self-Describing: Also related to the previous topic, a self-describing data model is a necessary characteristic derived from existing semi-structured data models that allows arbitrary data to be merged and exchanged without the need for domain specialists. This requirement allows us to arbitrarily combine information about the same object, where object identity can be determined by global identifiers.

Graph-Based: The need for a graph-based data model is necessary for the data integration step. If we focus on the data models we presented so far, we rapidly come to the conclusion that we need a graph-based data model: RDF is in itself graph-based and XML, although naturally a tree-based model, includes forms of graph representation (by means of giving XML nodes identifiers and then referring to them). As for the relational model, although it consists of a set of relations, the schema is actually a graph when we take into account foreign key constraints between different relations. When we take a schema-less data-model as the target data model, the schema information needs to be encoded in the data and as such, even for the relational model, we require a target data model that is a graph.

As we have seen, RDF has clear advantages over the relational and XML formats as a representation model for integrated data: (i) RDF is *per se* schema-unaware; (ii) by relying on URIs, it uses a standard mechanism for providing global identifiers for entities; and (iii) it is self-describing, since according to the LOD principles, by accessing each URI we obtain further information about the resource or by using RDF Schema (Brickley and Guha, 2004) we can further deduce implicit information.

1.3. Hypothesis

In this thesis we propose the use of RDF as a representation model for integrated data and extend RDF with support for meta-information, thus allowing us to deal with temporal and uncertain data. Several data models suitable for representing integrated data have been presented before, mostly tree or graph-based (Cluet et al., 1998; Papakonstantinou et al., 1995; Abiteboul, 1997), and we consider that RDF, being graph-based, is a well suited format for representing integrated data. We are particularly focusing on the conversion of data stored in legacy models, such as relational databases and XML, into RDF. The objective is to facilitate this data integration process by providing a query language that is capable of accessing the data stored in different source formats and thus, as opposed to traditional query languages, avoid the explicit need for *a priori* data translation, while allowing the target RDF data to be created.

Support for meta-information in RDF is necessary not only for representing integrated data, but also if the original source already contains some form of meta-information, such as temporal data, or access

restrictions. To represent temporal, fuzzy, provenance, or access control information, we need to consider an extension of the RDF data model that caters for such kinds of meta-information and further make the query language aware of this extension.

The main hypothesis of this thesis can be summarised as follows:

> **Efficient data integration over heterogeneous data sources can be achieved by: (i) a query language that allows to access data adhering to different formats in the original sources (without the need for data transformation); (ii) a set of optimisations that allow for efficient query evaluation in such a query language; and (iii) an interchange representation format based on RDF with support for meta-information, allowing to represent temporal, uncertain, provenance, or even access-control information.**

The proposed query language must be expressive enough to represent arbitrary transformations between data models, an important characteristic since one of the considered data models is the tree-based XML, where the merging of XML data needs to based on tree transformations.

When representing the integrated data as RDF extended with meta-information, other issues arise: how should RDFS handle such meta-information in the inference process? Or how can we query the meta-information? Extending the RDF data model with meta-information also requires a similar extension to SPARQL: the World Wide Web Consortium (W3C) recommended query language for RDF.

1.4. Contributions

We validate our hypothesis by designing a query language, called XSPARQL, that combines the XQuery, SPARQL, and the Structured Query Language (SQL) query languages, thus providing a cross-model query language suitable for data integration. The initial proposal of combining the XQuery and SPARQL query languages was presented before the start of this thesis by Akhtar et al. (2008). Shortly after this initial paper, I joined the project and started working on the following aspects, which constitute a substantial part of the presented thesis:

- formalising the existing XSPARQL language and extending its semantics to cater for both the XML and relational models;
- a set of optimisations over this novel language, specifically targeted at nested queries, that improve the evaluation times for such types of queries, including both formal proof of the correctness and empirical proof of the efficiency of such optimisations; and
- a general extension of the RDF data model, called Annotated RDFS, that allows to represent meta-information and forms the target data model for integrated data. We also detail extensions of the RDFS inference rules and the SPARQL query language that allow to infer new information and query this novel data model.

The difference in ordering of the data models is bridged at the query language level: even for the unordered data models (relational and RDF) their query languages (SQL and SPARQL, respectively) impose an ordering on the query results.[8] The XSPARQL query language is based on XQuery, which is an intrinsically ordered query language for XML data, and we rely on the implicit ordering provided the SQL and SPARQL query languages to maintain an ordered query language. For the purposes of data integration, where we are interested in generating RDF, the ordering in the query language is not important (since the target data model is unordered).[9]

[8] This is further detailed in Chapter 3.

[9] In Chapter 5 we exploit features of the XQuery language (and thus inherited by XSPARQL) that allow to disregard ordering during query evaluation.

1.4.1. Impact

For tackling our hypothesis points (i) and (ii), the consolidated work on the XSPARQL language, catering for the integration of the XML and RDF formats has been published in the Journal on Data Semantics (Bischof et al., 2012). This work formalises XSPARQL and introduces the optimisations for nested SPARQL queries in our current implementation of the XSPARQL language. Further expanding on our hypothesis point (i), the extension of XSPARQL to also support relational databases (Lopes, Bischof, Decker et al., 2011) was presented at the Portuguese Conference on Artificial Intelligence (EPIA2011).

Regarding our novel data model, hypothesis point (iii), we introduced the initial Annotated RDFS framework, along with the definition of the Fuzzy and Temporal annotation domains, was accepted to the AAAI Conference on Artificial Intelligence (AAAI 2010) (Straccia et al., 2010). Lopes, Polleres et al. (2010) later introduced the extension of the SPARQL query language that caters for querying domain annotations. This work was presented at the International Semantic Web Conference (ISWC-10). The consolidation of the Annotated RDFS model and the AnQL query language (Zimmermann et al., 2012) was published in the Journal of Web Semantics.

Finally, based on the data model proposed in this thesis, Lopes, Kirrane et al. (2012) specialises Annotated RDFS to the access control domain. This is also an important aspect when considering data integration since the underlying sources often to have their data protected by some form of access control. Using the combined XSPARQL query language, it is possible to extract the data and access control information from the underlying sources and replicate it as Annotated RDFS. This work has been accepted to the International Conference on Logic Programming.

1.4.2. Other Contributions

Since the focus of this thesis is on Web languages and data models, in addition to research publications, another important aspect of the dissemination of our results is the impact regarding standardisation. As such, parts of the work developed for this thesis have been submitted to the W3C in the form of Member Submissions or at the W3C organised workshop on RDF Next Steps: the first contribution was a W3C Member Submission describing the XSPARQL language. The aim of such Member Submissions is to make the W3C aware of technology being developed, which may be considered as input to future working groups. The XSPARQL W3C Member Submission was composed of four documents: (i) XSPARQL Language Specification (Polleres et al., 2009); (ii) XSPARQL: Implementation and Test-cases (Lopes, Krennwallner et al., 2009); (iii) XSPARQL: Semantics (Krennwallner et al., 2009); and (iv) XSPARQL: Use cases (Passant et al., 2009).

Two position papers were accepted to the W3C Workshop on RDF Next Steps. The purpose of this workshop was to gather feedback on possible improvements (if any) for the next iteration of the RDF language. The accepted position papers argued for the need to integrate XML and RDF by means of a query language (Lopes, Bischof, Erling et al., 2010), largely inspired by the XSPARQL language, and the need to cater for meta-information in RDF (Lopes, Zimmermann et al., 2010), calling for a framework similar to Annotated RDFS.

A presentation detailing the XSPARQL language and focusing on the integration of heterogeneous sources on the Web, such as XML and RDF in the form of Linked Data, was presented at the 2011 Semantic Technology (SemTech) Conference (Lopes and Polleres, 2011).

Also, my participation in the W3C RDB2RDF working group, currently a W3C recommendation, resulted in an implementation of the RDB2RDF Direct Mapping (Arenas, Prud'hommeaux et al., 2012) and R2RML (Das, Sundara et al., 2012) language specifications, using the XSPARQL language described in this thesis. The implementation of these specifications in XSPARQL was submitted to the W3C RDB2RDF Working Group.

1.5. Thesis Outline

Next we present an overview of each of the following chapters in this thesis:

Chapter 2 (Data Models) presents the necessary background information regarding the relevant data models considered in the integrated query language: the relational, XML, and RDF data models.

Chapter 3 (Query Languages) gives an overview of the query languages that can be used over the different data models: SQL for relational databases, XQuery for XML data, and SPARQL for data adhering to the RDF model.

Chapter 4 (The XSPARQL Language) introduces our combined query language, called XSPARQL, that allows data adhering to the relational, XML, and RDF data models to queried using a common language. The XSPARQL language consists of an extension of the XQuery query language with syntactical constructs taken from SQL and SPARQL. We present the syntax and semantics of XSPARQL, based on extending the XQuery formal semantics, and show correspondences between this novel query language and its composing languages.

Chapter 5 (XSPARQL Evaluation and Optimisations) describes our current implementation of the XSPARQL language, presents the experimental evaluation and some possible optimisations for XSPARQL queries. These optimisations focus on the interface between the different data formats, which in the case of nested queries, may cause severe evaluation overhead when compared with their single data model counterparts. We present an evaluation of the proposed optimisations based on a newly defined benchmark suite encompassing different data models.

Chapter 6 (An Extension of RDF and SPARQL towards Meta-Information) presents a common extension of the RDF data model, called Annotated RDFS, that caters for different kinds of meta-information, and facilitates the modelling of Temporal, Fuzzy, and Provenance meta-information in RDF. This chapter also includes the extension of the SPARQL query language to query the RDF annotated with meta-information. This extension of the SPARQL language, called AnQL, allows the user to write meta-information aware queries and we extend the SPARQL algebra to allow for the propagation of this meta-information in the query.

Chapter 7 (A Secure RDF Data Integration Framework) illustrates an integrated use case where the XSPARQL language further extended to support Annotated RDFS is used to extract legacy information contained in several enterprise systems and convert it to RDF while maintaining any existing access control permissions. We give an overview of a system architecture that, based on XSPARQL, extracts the data (along with the access control information) from the original sources, converts this data into Annotated RDFS, and enforces the access control permissions.

Chapter 8 (Conclusions) contains critical discussion of the presented work, highlights future directions of research and finishes with some concluding remarks.

Part I.

State of the Art

2. Data Models

This chapter details how data is represented in each of the data models mentioned in the previous chapter. In the context of databases, a common definition for the term *data model* is presented by Silberschatz et al. (1996) as "a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities". This definition focuses on the (essential) data representation capabilities of a data model; however more fine-grained definitions, namely by Codd (1980), also include in the notion of data model operators and inference rules for retrieving or deriving data as well as integrity rules for determining accepted database states. In this chapter, we consider the definition by Silberschatz et al. (1996) and are thus particularly interested in the data representation aspects of each data model: the relational model, the tree based data models XML and JSON, and RDF. However, we do touch upon the inferencing capabilities of RDF in Section 2.4.3 with RDF Schema; these will be required later in Chapter 6. In Chapter 3, we will focus on querying the presented data models.

Running example

In this thesis we will use examples from the music domain, where we are interested in representing *persons*, *bands*, *albums*, and *songs*. In our simplified model, persons can be members of bands and can listen to specific songs. Bands release albums, which in turn include songs. For presentation purposes and conciseness of examples, we will use a reduced set of entities, included in Example 2.1.

We chose to use the music domain, as opposed to more enterprise oriented examples, due to the availability of information. We note that data required for this use case is available in the WWW, for instance, information regarding bands can be found in Wikipedia (`http://www.wikipedia.org/`) or MusicBrainz (Swartz, 2002), while personalised information about the songs individuals listen to can be found in Last.fm (`http://last.fm/`).

Wikipedia/DBpedia. The widely known online encyclopaedia Wikipedia relies on user contributions for its contents. DBpedia (Bizer, Lehmann et al., 2009) consists of a partial export of the information from Wikipedia into the RDF format, accessible using standard query languages such as SPARQL. For our running example, we are interested in extracting information regarding artists, bands, and albums and we often use DBpedia URIs as identifiers for entities.

Last.fm. The online Web service Last.fm allows users to submit the songs and artists they listen to. These songs are aggregated in order to create a user profile containing the top artists, lists of songs from each user, and provide personalised recommendations of new artists. The data presented in this thesis was extracted from this author's Last.fm website.[1] The data retrievable via the Last.fm API contains information such as the five most played bands from a user profile and, for each band, the most played tracks by the user and the albums they are included in.

Example 2.1 (Use case data)**.** This example presents the data we are using in the examples of this thesis:[a]

persons: Marco Hietala, Tarja Turunen

[1] Last.fm user profile available at `http://last.fm/user/jacktrades/`, retrieved on 2012/04/10.

bands: Nightwish
albums: Wishmaster
songs: FantasMic, Wishmaster

In this thesis, we are interested in representing this data in four different formats: (i) a relational database, (ii) XML, (iii) RDF, and (iv) JSON. The representation of this data in each specific syntax is accompanied by the description of each data model throughout this chapter.

[a] For the sake of conciseness of examples, we restrict the presented data to one band, two members and one album of the band.

In the next sections we describe the relational model, the tree-based XML and JSON models, and the graph-based RDF data model. We conclude with an high-level comparison of the presented data models in Section 2.5. In Chapter 3, we will describe the respective query languages associated with each data model.

2.1. Relational Model

Due to the ever-growing need to store information, database systems were one of the most researched software systems and have evolved from the use of the filesystem to store the data into the currently ubiquitous relational database management systems (Abiteboul, Hull et al., 1995). Initially, the simple use of filesystems to store data did not enforce any structure on the data, where each file could have its own internal structure. One major turning point in the evolution of database systems was the separation of the *logical* definition of the data from its *physical* representation (known as the *data independence principle*). Thus, the task of managing the physical representation is left up to the *database management system* and is usually hidden from the database user. This separation also led to the development of several logical data models that allowed data to be described independently of their physical representation. The logical data models can be composed primarily of a Data Definition Language (DDL) and a Data Manipulation Language (DML). The DDL specifies the structure used to represent data while the DML specifies methods to access and update data. The hierarchical and network data models were the first logical models to be introduced, where the former used a tree structure for representing its data and the latter a graph structure. However, according to Abiteboul, Hull et al. (1995), major issues with these logical models were: (i) they were still closely related to the physical representation model; and (ii) their DML were limited, focusing mostly on navigating the physical representation.

The introduction of the relational model by Codd (1970), with its strong theoretical foundations, propelled database management systems forward, allowing for advances in efficient query translation methods (from the relational logical model into the physical representation model) and query optimisation techniques. In the relational model, data is represented primarily using named *relations*, where each *relational tuple* (or record) consists of several typed and named *attributes*. A commonly used alternative representation for relational data depicts each relation as a table, where the attributes are the columns of this table, and each relational tuple is represented as a row in the table. Next we present a definition of the relational model, based on Abiteboul, Hull et al. (1995), that relies on the pairwise disjoint and countably infinite sets $\mathbf{R}$ for relation names, $\mathbf{A}$ for attribute names and $\mathbf{D}$ for the domain of values that the attributes can hold. An element $d \in \mathbf{D}$ is called a *constant* and for an attribute $a \in \mathbf{A}$ we represent the domain of a as $dom(a)$. Furthermore, a total order is assumed between the elements of $\mathbf{A}$: this is a necessary feature to later allow us to specify relational instances in a similar fashion to logic programming (Lloyd, 1987).

Definition 2.1 (Relation and database schema)**.** *A* relation schema *is represented as* $r[U]$*, where* $r \in \mathbf{R}$ *is a relation name and* $U \subset \mathbf{A}$ *is a set of attribute names, called the* sort *of* r *and denoted by* $sort(r)$*. The*

arity *of* r *consists of its number of attributes:* $|sort(r)|$. *In turn, a* database schema S *is a non-empty and finite set of relation schemas.*

Example 2.2 (Relational Schema)**.** A possible schema for a relational database that stores information relevant to our use case is $S = \{ person, band, album, song \}$, where

$$\begin{aligned} sort(person) &= \{ personId, personName, bandId \} \\ sort(band) &= \{ bandId, bandName \} \\ sort(album) &= \{ albumId, albumName, bandId \} \\ sort(song) &= \{ songId, songName, albumId \} \end{aligned}$$

Other features of the relational model include primary and foreign keys. Intuitively, a primary key consists of a set of attributes that uniquely identify the tuples of a relation. For example, in our database schema we assume an artificially generated number that uniquely identifies each person or band (*personId* and *bandId*, respectively). Foreign keys are used to specify dependencies between attributes of two different relations: the connected attributes must have the same value in both relations. This can be seen in Example 2.2, where the same attribute names are used in different relations to specify the foreign keys, e.g. *bandId* in the relations *person* and *band*.

Furthermore, the null value is assumed to belong to all domains and, unless otherwise specified by means of constraints, can be used in place of any valid value for an attribute of a relation. The intended meaning of null values is to represent missing or unknown information. However, since null values greatly complicate the definition of the algebra operations (presented in Section 3.1.2), we will, for the most part, ignore null values in the presented definitions.

Database Instances

Abiteboul, Hull et al. (1995) present different perspectives for representing relational tuples i.e. *instances* of relational schemas, the *conventional* and *logic programming* perspectives. The so-called *conventional* perspective on relational databases (used later in Section 3.1.2) represents tuples as functions, where a tuple t over a finite set of attributes U consists of a function u with domain U. The sort of u is U and the value of u of an attribute $a \in U$ is denoted $u(a)$. Extending this notion to a set of attributes $V \subseteq U$, we say that $u[V] = u|_V$ denotes the restriction of the function u to V, i.e. $u[V]$ denotes a new tuple v over V such that $v(a) = u(a)$ for each attribute $a \in V$.

An alternate view focuses on the *logic programming* perspective, under which a relational tuple can be viewed as a *fact*. For a relation name r with arity n, a fact is an expression $r(a_1, \ldots, a_n)$, where each $a_i \in \mathbf{D}$ is a constant. Facts can also be represented as $r(u)$, where $u = \langle a_1, \ldots, a_n \rangle$. According to this representation, a *relation instance* over a relation schema r is a finite set of facts over r and a *database instance* over a database schema S is the union of all relation instances over r, for each relation schema $r \in S$. Since relations are represented as sets, the standard set operations of intersection, union and difference ($\cap, \cup$, and $-$, respectively) can be applied and relations can be compared using the $\subset, \subseteq, =$, and $\neq$ operators.[2] Example 2.3 represents a database instance following the logic programming perspective.

Example 2.3 (Database Instance)**.** The database instance containing the use case data from Example 2.1, over the database schema presented in Example 2.2, is as follows:

[2]We note that although the relational model is formally described using a set based semantics, it is common for database systems to use multi-sets for representing the data and the results of SQL queries.

$$\{\ person(1, \textit{'Marco Hietala'}, 1)\,, person(2, \textit{'Tarja Turunen'}, 1)\,,$$
$$band(1, \textit{'Nightwish'})\,,$$
$$album(1, \textit{'Wishmaster'}, 1)\,,$$
$$song(1, \textit{'FantasMic'}, 1)\,, song(2, \textit{'Wishmaster'}, 1)\}$$

Both views on relational data are equivalent and are used for different formalisations of the relational model and query languages (as presented in Section 3.1).

2.2. Extensible Markup Language (XML)

As we have highlighted in Chapter 1, with the growing success of the WWW, where data exposed as HTML is often extracted from relational databases, the need to query Web Data in a structured way and thus consider the Web as a *global database* increased (Silberschatz et al., 2005; Abiteboul, Buneman et al., 1999). Also powered by several data integration projects, research began to focus on the representation and querying of semi-structured data following a *graph* or *tree* structure. Semi-structured data models were devised as the required formats for representing data available on the Web and as a representation-independent way to transfer data between different database management systems (Abiteboul, 1997; Buneman, 1997).

The Extensible Markup Language (XML) (Bray, Paoli, Sperberg-Mcqueen et al., 2008) is a semi-structured representation format and, with the support of the W3C, it has become the *de facto* standard for data exchange on the Web (Suciu, 1998; Abiteboul, Buneman et al., 1999). XML is a subset of the Standard Generalized Markup Language (SGML) ISO standard (ISO, 1986) and is designed to be compatible with SGML and HTML. XML represents data in a tree-like format that, when compared to the relational format, is a more flexible data representation format and is also considered easier to read and write for both humans and machines.

XML has also brought forward a new class of databases: *XML databases*. Although currently most databases provide easy creation of XML data, for example by exporting the data they contain as an XML document, XML databases refer to a database management system that manage collections of XML data (Katz et al., 2003). Even though the data may be internally represented in another format, access and manipulation is based on XML formats and languages.

Data 2.1 contains the representation of the use case data from Example 2.1 in XML. This document starts by representing a user and the top bands they listen to, where each band includes information regarding its members and albums, and for each album, the songs listened to by the user. As per Bray, Paoli, Sperberg-Mcqueen et al. (2008), the Extensible Markup Language describes what are called *XML documents*, which are composed primarily of XML *elements*. In turn, XML elements consist of a *start-tag*, the *element content*, and an *end-tag*. Consider the following XML element:

```
<song>Wishmaster</song>
```

Start- and end-tags are indicated by "`<song>`" and "`</song>`", respectively, where "`song`" is called the *element name*, and the element content may consist of text (any string of characters), other (nested) XML elements, CDATA sections, processing instructions or comments. CDATA sections can be used to include text that contains markup characters (such as the start- and end-delimiters) and processing instructions contain data that is to be sent to the application consuming the document. Comments can be present anywhere in the document and, similar to any programming language, can be ignored by the XML processor. Furthermore, XML elements may contain *attributes* enclosed in their start-tags, in this case the "`album`" element has the "`name`" attribute with value "`Wishmaster`":

```
<album name="Wishmaster">
```

```xml
<?xml version="1.0"?>
<user>
  <bands>
    <band name="Nightwish">
      <members>
        <member>Marco Hietala</member>
        <member>Tarja Turunen</member>
      </members>
      <albums>
        <album name="Wishmaster">
          <song>FantasMic</song>
          <song>Wishmaster</song>
        </album>
      </albums>
    </band>
  </bands>
</user>
```

Data 2.1: Bands in XML (*bands.xml*)

In XML text, elements, CDATA, processing instructions, comments, and attributes are collectively referred to as *XML nodes*.

2.2.1. XML Namespaces

XML provides a way to disambiguate entities such as element and attribute names by using XML namespaces (Bray, Hollander et al., 2009), where each XML namespace is identified by a URI reference (Berners-Lee, Fielding et al., 2005). XML allows, by means of reserved attributes, to associate partial URIs with a *prefix name* and/or to declare a *default namespace*. *Qualified names* (or QNames) provide a convenient form of naming element and attribute names in XML and can be composed of prefixed or unprefixed names. Prefixed names make use of the previously declared prefixes and are combined with the *local part* to specify the URI reference. For unprefixed names, if a default namespace declaration exists it is taken as the namespace value, otherwise there will be no namespace value. For example, including the "`xmlns`" attribute in an XML element declares the default namespace to be used within that element:

```xml
<user xmlns="http://example.org/bands/">
```

while URIs can be associated with a prefix in the following manner:

```xml
<members xmlns:foaf="http://xmlns.com/foaf/0.1/">
```

XML namespaces are scoped to the element in which they are declared, including any child elements.

2.2.2. XML Validation

The XML W3C specification (Bray, Paoli, Sperberg-Mcqueen et al., 2008) defines two levels of conformance for XML documents: *well-formed* documents and *valid* documents. *Well-formedness* constraints primarily ensure that the XML document follows syntactic specifications, such as (to name but a few): (i) they must contain at least one element; (ii) a distinct element, called the *root*, is not included in the content of any other element; (iii) for all non-root elements, its start- and end-tags must be included within the content of the same element, i.e. opening and closing tags must not overlap; and (iv) attribute names must be unique within the same element.

On the other hand, *valid* documents rely on a *schema* that, similar to relational databases, specifies the structure of a particular class of XML documents. Such schemas can be specified using two different

```
<!DOCTYPE user [
  <!ELEMENT user (bands)>
    <!ATTLIST user username CDATA #REQUIRED>
  <!ELEMENT bands (band*)>
  <!ELEMENT band (members,albums)>
    <!ATTLIST band name CDATA #REQUIRED>
  <!ELEMENT members (member*)>
  <!ELEMENT member (#PCDATA)>
  <!ELEMENT albums (album*)>
  <!ELEMENT album (song*)>
    <!ATTLIST album name CDATA #REQUIRED>
  <!ELEMENT song (#PCDATA)>
]>
```

Figure 2.1.: DTD definition for the bands XML data

formats: Document Type Definition (DTD) or XML Schema, both of which are detailed below.[3] In Chapter 4 we will define XML Schema datatypes for representing RDF concepts and thus incorporating them into XQuery.

Document Type Definition

DTD specifications are mostly referenced here for historical reasons, since XML Schema is more widely used (as detailed in the next section). DTD specifications consist of *markup declarations*, such as *element type*, *attribute list*, *entity* or *notation* declarations. Element type declarations are defined using the '`ELEMENT`' keyword, for instance:

```
<!ELEMENT album (song*)>
```

specifies an "`album`" element that is constituted by any number of "`song`" elements. The "`album`" element is required to have an attribute named "`name`" by the following attribute list declaration:

```
<!ATTLIST album name CDATA #REQUIRED>
```

The complete DTD definition for the use case XML structure is presented in Figure 2.1. An attribute declared as `CDATA` indicates that its value must be a sequence of characters and/or XML markup. On the other hand, `PCDATA` (meaning "parsed character data") indicates that only one text element, and no other nodes are allowed in the content. Adding this DTD definition to the XML document from Data 2.1 would ensure that any validating *XML processor* checks the structure of the XML data against the provided schema definition.

XML Schema

While DTDs are included in the W3C XML specification and therefore are widely available, there are some drawbacks to their use, most noticeably the lack of namespace support. To overcome such drawbacks, the W3C has defined the XML Schema specification, composed of two parts: (i) an XML-based syntax for validating XML documents (Thompson et al., 2004); and (ii) a specification of XML datatypes (Biron and Malhotra, 2004).

The XML Schema definition of the use case XML data is presented in Figure 2.2, which has the same effect as the DTD presented in Figure 2.1: validating the XML document from Data 2.1. In XML Schema, XML elements and attributes are declared using an XML element named "`element`" and "`attribute`",

[3] There are other schema languages for XML, such as the Relax NG language, but for the scope of this thesis we will focus on W3C specifications.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="user">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="bands"/>
      </xs:sequence>
    <xs:attribute name="username" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="bands">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="band" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="band">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="members"/>
        <xs:element ref="albums"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="members">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="member" type="xs:string" minOccurs="0" maxOccurs="unbounded"/
            >
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="albums">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="album" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="album">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="song" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.2.: XML Schema definition for Bands XML data (partial)

respectively, contained in the "`http://www.w3.org/2001/XMLSchema`" namespace. For example, the "`album`", along with its "`name`" attribute and "`song`" elements, are defined in lines 41–48 of Figure 2.2.

The specification of datatypes in XML (Thompson et al., 2004) introduces a datatype system that is also used by other W3C specifications, such as the RDF specification (Manola and Miller, 2004). A datatype is defined by (a) the *value space*: a set of values for a datatype; (b) the *lexical space*: a set of valid character strings for the datatype; and (c) a *lexical-to-value mapping* linking elements of these two sets. A datatype is identified by a URI and a *datatype map* associates the URI with the specific datatype. The defined datatype system allows for the creation of user-defined datatypes, where such datatypes are *derived* from existing datatypes (called the *base type*) by restricting or extending its value space and lexical space.

The formalisation of XML Schema was proposed by Siméon and Wadler (2003), where the authors also describe a more human readable notation for both XML elements and XML schema types. This notation was later adopted by the XQuery semantics specification (Draper, Fankhauser et al., 2010). Following this notation, the XML element `<song>Wishmaster</song>` is represented as `element song { "Wishmaster" }`. The "`song`" and "`album`" elements from the XML Schema in Figure 2.2 can be represented in the shorthand notation as:

```
define element song of type xs:string
define element album of type albumType
define type albumType {
  element song*,
  attribute name of type xs:string }
```

After performing validation with the presented XML Schema, the XML element is represented as: `element song of type xs:string { "Wishmaster" }`. In Chapter 4 we specify the types introduced by the XSPARQL language following this notation.

2.2.3. XML Abstract Representations

The W3C specifications are defined over abstract representations of XML documents, with the objective of omitting the concrete syntax of XML documents, namely the XML Information Set (Infoset) and the XQuery 1.0 and XPath 2.0 Data Model (XDM). The Infoset provides the basic definitions for describing well-formed XML documents, with the purpose of serving as a reference for other XML specifications. On the other hand, the more complex XDM is meant to act as a data model for the XPath, XSLT and XQuery languages: describing their input documents and the values for expressions. These query languages will be the focus of Section 3.2. The Infoset describes only the basic information contained in an XML document, while the XQuery and XPath Data Model is used mostly for the XML Query languages (described in Section 3.2).

XML Infoset

The XML Information Set, described by Cowan and Tobin (2004), provides definitions referring to a well-formed XML document and as such, any well-formed XML document, although not necessarily a valid document, will have an Infoset. The Infoset consists of a set of *information items*, where each information item describes a part of the XML document by means of *properties* that refer to other information items.

An Infoset contains exactly one document information item that, directly or indirectly, refers to all of the other information items in the set. Other information items are used to represent XML nodes such as elements, attributes, processing instructions, or comments.

XQuery 1.0 and XPath 2.0 Data Model (XDM)

The XPath, XSLT, and XQuery languages use the *XQuery 1.0 and XPath 2.0 Data Model (XDM)* (Fernández et al., 2010) for describing their input XML documents. XDM is based on the XML Infoset and extends it with support for: (i) XML Schema types (Thompson et al., 2004; Biron and Malhotra, 2004); (ii) typed atomic values; (iii) collections of documents and complex values; and (iv) ordered, heterogeneous sequences.

The basic element of the data model is called an *item*, which has a type (either an XML *node* or an atomic type) and an associated value. Types in the data model are uniquely represented using (expanded) QNames and the pre-defined atomic types include the built-in types presented in Biron and Malhotra (2004), extended with the following additional types: (i) `xs:anyAtomicType`; (ii) `xs:untyped`; (iii) `xs:untypedAtomic`; (iv) `xs:dayTimeDuration`; and (v) `xs:yearMonthDuration`.

The data model defines seven types of XML nodes, namely: document, element, text, attribute, namespace, processing instruction, and comment. Each item that represents an XML node is associated with the corresponding type and for every type of node, it is possible to compute a string value. In the data model, each XML node has a unique identity, which is equal only to itself. On the other hand, all instances of the same atomic values are considered equal, i.e. they do not have a unique identity. XDM defines a total order among all available nodes, called *document order*, which consists of the order that nodes appear in the original document.

In comparison to the Infoset, another addition of the XDM is the support for sequences. A *sequence* consists of any number of items (XML nodes and/or atomic values) and are represented as a comma-separated '`,`' list of items, delimited by the '`(`' and '`)`' characters. Each item is considered equal to a singleton sequence containing that item and furthermore sequences are not allowed to include any other sequences and are thus always considered as a flattened sequence. For example, the sequence "`(1,(<a/>), "3")`" is translated to "`(1,<a/>, "3")`".

2.3. JavaScript Object Notation (JSON)

JSON is defined as a "lightweight data-interchange format" is another tree-based model designed as an alternative to XML for data transmission between different applications. Although it originates from the JavaScript language, its format is language independent and thus can be used by several programming languages.

The main structure of JSON is called an *object*, enclosed between '`{`' and '`}`', and consists of an unordered sequence of *name-value* pairs, separated by '`:`'. In such structures, the *name* is restricted to be a string while *value* may be one of (a) string; (b) number; (c) object; (d) array; (e) boolean (true or false); or (f) null. *Arrays* consist of an ordered list of values and are enclosed between '`[`' and '`]`'. The JSON representation of our use case data is presented in Data 2.2. This simple and unambiguous structure, coupled with the fact that JSON is natively recognised and imported by JavaScript, made JSON extremely popular on the Web. A comparative study of the uptake of XML and JSON was presented by Musser (2011).[4]

Although JSON and XML serve very similar purposes, commonly presented advantages for using JSON over XML are: (i) JSON documents are (usually) smaller; and (ii) an external schema is not required to unambiguously represent the content. On the other hand, one of the biggest disadvantages of JSON is its lack of support for namespaces: whereas in XML it is possible to distinguish attribute and element names by giving them different namespace prefixes, this is not possible in JSON.

[4]The presentation is available at `http://www.slideshare.net/jmusser/j-musser-semtechjun2011/`, retrieved on 2012/04/10.

```
{
    "bands": {
        "Nightwish": {
            "members": [
                "Tuomas Holopainen",
                "Tarja Turunen"
            ],
            "albums": {
                "Wishmaster": [
                    "Wishmaster",
                    "FantasMic"
                ]
            }
        }
    }
}
```

Data 2.2: Bands in JSON (*bands.json*)

Due to the similarities between the JSON and XML formats, the question of translating between them has arisen. Since XML is arguably the more expressive language, being more flexible in its format, converting from XML to JSON poses some problems:

Namespaces. Since JSON does not natively support namespaces, a non-trivial issue is how to represent XML namespaces in such a fashion that can be translated back into XML;

Attributes. Similar to namespaces, JSON has no equivalent for XML element attributes and similar representational questions arise for XML attributes;

Mixed Content. Since the contents of XML elements can consist of text values arbitrarily mixed with other elements, an accurate representation of this mixed content in JSON, although possible, would yield a very verbose representation.

On the other hand, converting from JSON to XML is a straightforward task, relying solely on using predefined element names to represent JSON objects and arrays. This straightforward conversion will be used in Chapter 4 for the inclusion of JSON data into our proposed transformation language.

2.4. Resource Description Framework (RDF)

In the attempts to transform the Web into a global database, another model, the Resource Description Framework, was proposed as the data model for representing machine readable data, also known as *Semantic Web* data. The RDF model allows for the specification of *statements* about *Web Resources* (Manola and Miller, 2004). However, this general notion of resource may refer not only to virtual entities (that can only be found on the Web) but also any physical entity that can be *identified* on the Web. Such resources are identified by a URI, generally indicating where the resource is located, or a *blank node*, which plays the role of an anonymous resource and allows for the modelling of incomplete or unknown data. In the following, we identify blank nodes by using the prefix '`_:`' followed by a string, called the *blank node label*. Blank nodes are scoped to the document they appear in, i.e. two blank nodes from different documents, even if they have the same label, must be considered different. Furthermore, RDF *literals* can be used to specify string- or datatype-based values for properties. The atomic *statements* of the RDF data model are called *RDF triples* consisting of *subject*, *predicate* and *object*, and intuitively state that the subject is connected to the object by the predicate relation. Since triples can also be viewed

as part of a labelled directed graph, where subjects and objects correspond to nodes and predicates to edges of the graph, we refer to a set of such RDF triples as an *RDF graph*.

For the definitions of the RDF syntax, we rely on the the pairwise disjoint alphabets **U**, **B**, and **L** denoting *URI references*, *blank nodes* and *literals*, respectively.[5] We call the elements in **UBL** *terms*.

Definition 2.2 (RDF Triple)**.** *An* RDF triple *is* $\tau = (s, p, o) \in \mathbf{UBL} \times \mathbf{U} \times \mathbf{UBL}$*, where* s *is called the* subject*,* p *the* predicate*, and* o *the* object*.*

Strictly speaking, according to the RDF specification (P. Hayes, 2004) literals are not allowed to be the subject of RDF triples however, as commonly adopted in other works (Muñoz et al., 2007; Prud'hommeaux and Seaborne, 2008; Carroll, Bizer et al., 2005), this definition considers a *generalised RDF Triple*, that allows literals for the subject positions.

Definition 2.3 (RDF Graph)**.** *Following the definition of an RDF triple, an* RDF graph G *consists of a set of triples. The universe of* G*,* $universe(G)$*, is the set of elements in* **UBL** *that occur in the triples of* G *and the* vocabulary *of* G*,* $voc(G)$*, is* $universe(G) \cap \mathbf{UL}$*. Furthermore, we say that* G *is* ground *if and only if* $universe(G) = voc(G)$*, i.e.* G *does not contain blank nodes.*

When combining different RDF graphs some care must be taken to ensure the local scope of blank nodes:

Definition 2.4 (RDF merge)**.** *Let* S *be a set of RDF graphs. The* RDF merge *of* S *consists of the set-theoretical union of all the graphs in* S *after blank nodes have been* standardised apart*: if any two graphs contain the same blank node label, all occurrences of these labels within the same graph are replaced by a new blank node label that is not present in any of the other graphs.*

This disambiguation of blank node labels is meant to keep any blank nodes between different graphs distinct, thus maintaining the scope of blank nodes to the graph they occur in.

Similar to XML namespaces, URIs can be abbreviated by using a namespace prefix. For example, the URI "`foaf`:`Person`" from the widely used Friend Of A Friend (FOAF) ontology, consists of the prefix "`foaf`", which is associated with the URI "`http://xmlns.com/foaf/0.1/`", and the local part "`Person`". The complete URI represented by "`foaf`:`Person`" is thus "`http://xmlns.com/foaf/0.1/Person`". `rdf`:`type` predicates can be used to specify that an RDF resource is an *instance* of a *class*; for example the triple:

$$(\texttt{dbpedia:Marco\_Hietala}, \texttt{rdf:type}, \texttt{foaf:Person}) \tag{2.1}$$

intuitively specifies that the resource `dbpedia`:`Marco_Hietala` is used to identify a person.

RDF literals can be further classified as *plain*, in which case they can optionally contain a *language tag*, or *typed* literals. Typed literals include a URI that refers to their datatype, usually one of the XML Schema built-in datatypes or the newly defined RDF datatype `rdf`:`XMLLiteral` (used to indicate the literal contains XML data). The specific syntax of literals is presented in the next section.

Another RDF feature, although not so commonly used, is *reification*, can be used to represent meta-information about an RDF triple, e.g. provenance information. Any RDF statement can be reified by representing it as four distinct RDF triples with a common subject. Although it is possible to use a URI for the subject of reified triples, as presented in Example 2.4, it is common to use a blank node. Reification is later used in Chapter 6 as one possible serialisation for Annotated RDFS graphs.

Example 2.4 (Reified RDF statement)**.** The RDF statement (2.1) can be reified as the following triples:

[5] We assume $\mathbf{U}, \mathbf{B}$, and $\mathbf{L}$ fixed, and for presentation purposes we will denote unions of these sets by concatenating their names.

(_:r1, rdf:type, rdf:Statement)

(_:r1, rdf:subject, dbpedia:Marco_Hietala)

(_:r1, rdf:predicate, rdf:type)

(_:r1, rdf:object, foaf:Person)

Yet another feature of RDF are *collections*, which allow to state that a group of resources are members of the collection. This is represented in RDF using a list structure following a predefined vocabulary: `rdf:List` states the type of the resource and the `rdf:first` and `rdf:rest` properties are used to represent the list. This list must be terminated by `rdf:nil`. Collections are used in Section 4.2.2.

Next, Section 2.4.1 presents how RDF can be serialised in order to be stored or exchanged, focusing on the RDF/XML and Turtle syntaxes and Section 2.4.2 presents the semantics of RDF. Finally, Section 2.4.3 focuses on the inferencing capabilities of RDF by describing RDF Schema.

2.4.1. Representation Syntaxes

Although the RDF specification states that the normative syntax for writing RDF graphs is RDF/XML (Beckett, 2004), this syntax is not favoured among practitioners and there have been proposals to support other serialisation formats and move away from XML based representations (Beckett, 2010). Other well known syntaxes for RDF are Turtle (Beckett and Berners-Lee, 2011) and RDFa (Adida and Birbeck, 2008), where Turtle consists of a specialised syntax for RDF and RDFa defines a mechanism to incorporate RDF statements into (X)HTML webpages. In the following, we briefly highlight the constructs of the RDF/XML and Turtle syntaxes.

RDF/XML

Although RDF/XML is the normative syntax for RDF, this serialisation is very flexible, and the same RDF graph can be serialised in numerous different ways. As we will point out in Chapter 4, this lack of a canonical RDF/XML serialisation is one of the major roadblocks to process RDF data using XML tools.

The RDF/XML syntax uses XML elements to represent RDF subjects, predicates and objects: "`rdf:Description`" elements are used to represent nodes (subjects and objects) of the RDF graph, where the "`rdf:about`" attribute specifies the URI of the node. In turn, predicates are represented as XML elements where the name of the element corresponds to the URI (represented as an XML QName) of the predicate. A possible RDF/XML serialisation of the RDF graph from our running example is presented in Data 2.3.

The RDF/XML serialisation allows the use of several abbreviations and an abbreviated serialisation of the RDF graph in Data 2.3 is presented in Data 2.4. One of the possible abbreviations is, if an object node does not contain any other predicates, to omit the "`rdf:Description`" element and specify the URI reference of the object in the "`rdf:resource`" attribute of the predicate element node (for example in lines 13 and 14). Another abbreviation is, in case the subject contains an "`rdf:type`" predicate, to replace the "`rdf:Description`" element name with the type of the subject (for example lines 9, 12, and 16). Also, several predicates about the same subject can be nested in the same XML element (as presented in lines 19–24 of Data 2.4).

Blank nodes (anonymous resources) can be given a label using the "`rdf:nodeID`" attribute and can later be referred to (from within the same document). Literals can be specified as the text content of a property XML element (e.g. line 17 of Data 2.4) or alternatively as the value of an attribute where the attribute name is the corresponding property URI (as in line 12 of Data 2.4). Language tags are specified as the value of the "`xml:lang`" attribute, while typed literals use the "`xml:datatype`" attribute.

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:dbpedia="http://dbpedia.org/resource/"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/bands#"
         xmlns:foaf="http://xmlns.com/foaf/0.1/"
         xmlns:mo="http://purl.org/ontology/mo/"
         xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description rdf:about="http://dbpedia.org/resource/Nightwish">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/Band"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Nightwish">
    <foaf:name>Nightwish</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Nightwish">
    <foaf:member rdf:resource="http://dbpedia.org/resource/Marco_Hietala"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Nightwish">
    <foaf:member rdf:resource="http://dbpedia.org/resource/Tarja_Turunen"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Marco_Hietala">
    <foaf:name xml:lang="en">Marco Hietala</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Marco_Hietala">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/MusicArtist"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Tarja_Turunen">
    <foaf:name xml:lang="en">Tarja Turunen</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about="http://dbpedia.org/resource/Tarja_Turunen">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/MusicArtist"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/bands#album208">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/Record"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/bands#album208">
    <mo:title>Wishmaster</mo:title>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/bands#album208">
    <foaf:maker rdf:resource="http://dbpedia.org/resource/Nightwish"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/bands#album208">
    <mo:track rdf:nodeID="song566"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.org/bands#album208">
    <mo:track rdf:nodeID="song506"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="song566">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/Track"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="song566"><dc:title>Wishmaster</dc:title></
      rdf:Description>
  <rdf:Description rdf:nodeID="song506">
    <rdf:type rdf:resource="http://purl.org/ontology/mo/Track"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="song506"><dc:title>FantasMic</dc:title></
      rdf:Description>
</rdf:RDF>
```

Data 2.3: Bands in RDF/XML

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF
   xmlns:dbpedia="http://dbpedia.org/resource/"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:ex="http://example.org/bands#"
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:mo="http://purl.org/ontology/mo/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <mo:MusicArtist rdf:about="http://dbpedia.org/resource/Marco_Hietala">
    <foaf:name xml:lang="en">Marco Hietala</foaf:name>
  </mo:MusicArtist>
  <mo:MusicGroup rdf:about="http://dbpedia.org/resource/Nightwish" foaf:name="
      Nightwish">
    <foaf:member rdf:resource="http://dbpedia.org/resource/Marco_Hietala"/>
    <foaf:member rdf:resource="http://dbpedia.org/resource/Tarja_Turunen"/>
  </mo:MusicGroup>
  <mo:MusicArtist rdf:about="http://dbpedia.org/resource/Tarja_Turunen">
    <foaf:name xml:lang="en">Tarja Turunen</foaf:name>
  </mo:MusicArtist>
  <mo:Record rdf:about="http://example.org/bands#album208">
    <mo:title>Wishmaster</mo:title>
    <mo:track rdf:nodeID="song506"/>
    <mo:track rdf:nodeID="song566"/>
    <foaf:maker rdf:resource="http://dbpedia.org/resource/Nightwish"/>
  </mo:Record>
  <mo:Track rdf:nodeID="song506">
    <dc:title>FantasMic</dc:title>
  </mo:Track>
  <mo:Track rdf:nodeID="song566">
    <dc:title>Wishmaster</dc:title>
  </mo:Track>
</rdf:RDF>
```

Data 2.4: Bands in abbreviated RDF/XML

Turtle

Stemming from its XML roots and, even with all the proposed abbreviations, the RDF/XML syntax is still very verbose and neither easy to read nor write for humans. To overcome these problems, the Turtle syntax (Beckett and Berners-Lee, 2011) aims to be a compact representation for RDF graphs that is easier to read and write for users and includes abbreviations for common RDF patterns. Turtle is based on N-Triples, a simple syntax introduced for the RDF test cases (Grant and Beckett, 2004) that represents one triple per line. Furthermore, Turtle incorporates features from Notation 3 (Berners-Lee, 2005), most notably: (i) namespace declarations, (ii) shortcuts for commonly used RDF patterns, and (iii) a syntax for anonymous blank nodes.

The Turtle RDF representation of the use case data from Example 2.1 is presented in Data 2.5. In the Turtle syntax, `@prefix` declarations can be used to abbreviate common URIs (similar to XML namespaces and QNames) and URIs must be enclosed between the '`<`' and '`>`' characters.

Literals are surrounded by double-quotes, as in `"Nightwish"`, and may include a suffix to specify the language tag following the '`@`' separator character, for example `"Marco Hietala"@en`, or a datatype URI after the '`^^`' separator as in `"5"^^<http://www.w3.org/2001/XMLSchema#integer>`. Blank nodes are prefixed with '`_:`' e.g. `_:song506` where the blank node label is `song506`. A shortcut for unnamed blank nodes is provided by using the '`[]`' notation. Another useful shortcut is the '`a`' keyword (line 7 of Data 2.5), which represents the URI `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`, also commonly abbreviated as

```
1  @prefix ex: <http://example.org/bands#> .
2  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3  @prefix dbpedia: <http://dbpedia.org/resource/> .
4  @prefix mo: <http://purl.org/ontology/mo/> .
5  @prefix dc: <http://purl.org/dc/elements/1.1/> .
6
7  dbpedia:Nightwish a mo:MusicGroup .
8  dbpedia:Nightwish foaf:name "Nightwish" .
9  dbpedia:Nightwish foaf:member dbpedia:Marco_Hietala .
10 dbpedia:Nightwish foaf:member dbpedia:Tarja_Turunen .
11 dbpedia:Marco_Hietala foaf:name "Marco Hietala"@en .
12 dbpedia:Marco_Hietala a mo:MusicArtist .
13 dbpedia:Tarja_Turunen foaf:name "Tarja Turunen"@en .
14 dbpedia:Tarja_Turunen a mo:MusicArtist .
15 ex:album208 a mo:Record .
16 ex:album208 mo:title "Wishmaster" .
17 ex:album208 foaf:maker dbpedia:Nightwish .
18 ex:album208 mo:track _:song566 .
19 ex:album208 mo:track _:song506 .
20 _:song566 a mo:Track .
21 _:song566 dc:title "Wishmaster" .
22 _:song506 a mo:Track .
23 _:song506 dc:title "FantasMic" .
```

Data 2.5: Bands in Turtle (*bands.ttl*)

`rdf:type`.

Furthermore the '`;`' and '`,`' symbols can be used to create new triples without repeating the subject or subject and predicate, respectively. For example, the triples from lines 7–10 of Data 2.5 can be written as:

```
dbpedia:Nightwish a mo:MusicGroup ;
                  foaf:name "Nightwish" ;
                  foaf:member dbpedia:Marco_Hietala, dbpedia:Tarja_Turunen .
```

Commonly used datatypes can also be abbreviated: for instance `5` is equivalent to `"5"^^<http://www.w3.org/2001/XMLSchema#integer>`, while `5.0` corresponds to `"5.0"^^<http://www.w3.org/2001/XMLSchema#decimal>`. Turtle also provides abbreviations for RDF collections by listing a space-separated sequence of RDF terms enclosed by '`(`' and '`)`'.

2.4.2. Semantics

The semantics of RDF is specified using a model theory as per P. Hayes (2004), which is a common form of specifying semantics, for example for first-order logic. Model theoretic semantics of formal languages assign any expression in the language to an element of a possible "world" – called an *interpretation* – and also specify the necessary conditions for an interpretation to be considered valid – called a *model*. The notion of *entailment* between two expressions, A entails B, can then be defined as any interpretation that is a model of A must also be a model of B. Based on this semantics, it is possible to define what are the *entailed* consequences of the interpretation and what are valid *inference rules*.

In the case of RDF, language expressions are considered as being the terms in the universe of the graph and also the individual triples, i.e. each term in the vocabulary is assigned to an interpretation element, where plain literals are interpreted as themselves and blank nodes are interpreted as existential variables (scoped to the RDF graph in which they occur).

The RDF semantics (P. Hayes, 2004) specifies different types of interpretation, and hence of entailment, ranging from the so-called *simple* interpretation to the more complex *RDFS* and *datatype* interpretations.

Simple interpretations consider only the vocabulary of the triples present in the graph while other types of interpretations, namely RDF and RDFS-interpretations, consider predefined vocabularies and a set of RDF triples that any interpretation must satisfy by default: the so-called *axiomatic* triples.

RDF interpretations consider the terms defined in the `http://www.w3.org/1999/02/22-rdf-syntax-ns#` namespace (commonly abbreviated with the prefix `rdf`). For instance, RDF interpretations impose conditions that identify a subset of the interpretation resources as being *properties* (interpretation resources of type `rdf:Property`) and introduce the new datatype `rdf:XMLLiteral` to represent well-formed XML literals.

RDFS-interpretations consider further vocabulary in the `rdfs` namespace (`http://www.w3.org/2000/01/rdf-schema#`) that define further conditions on top of RDF interpretations and introduce the notion of a *class*. A class is itself a resource that denotes a common set of resources, which are called *instances* of the class and all have the class as the value for their `rdf:type` property. Informally, the RDFS vocabulary states the following: (i) $(p, \texttt{rdfs:subPropertyOf}, q)$ means that any resources related by property p are also related by property q; (ii) $(c, \texttt{rdfs:subClassOf}, d)$ means that any instance of class c is also an instance of class d; (iii) $(a, \texttt{rdf:type}, c)$ means that a is an instance of c; (iv) $(p, \texttt{rdfs:domain}, c)$ means that the *domain* of property p is c, i.e. any resource that is the subject of a triple with predicate p is an instance of c; and (v) $(p, \texttt{rdfs:range}, c)$ means that the *range* of property p is c, i.e. any resource that is an object of a triple with predicate p is an instance of c.

Further extending RDFS-interpretations, a D-interpretation provides an (admittedly minimal) support for XML Schema (XSD) datatypes (Biron and Malhotra, 2004) extended with the `rdf:XMLLiteral` datatype. Another W3C specification that provides a more expressive inference system than RDFS is the Web Ontology Language (OWL), now in its second version (Hitzler et al., 2009). The OWL language introduces new concepts such as the distinction between object and datatype properties, class disjointness assertions, and assertions of equality between individuals, among others. It is noteworthy that D-interpretations, OWL, and the `rdf:XMLLiteral` datatype may introduce inconsistencies in RDF. However, in this thesis, we are mostly interested in RDFS inferences and we do not consider D-interpretations, OWL constructs, nor the typing of `rdf:XMLLiteral` and thus we avoid any inconsistencies in RDF.

Although in the RDF Semantics specification (P. Hayes, 2004) the semantic conditions for each interpretation are detailed separately, in this thesis we follow the formalism defined by Muñoz et al. (2007); Muñoz et al. (2009) and provide a single notion of interpretation that covers Simple, RDF, and RDFS-interpretations. Intuitively, the interpretation of an RDF triple (s, p, o) is true if s, p and o belong to the interpretation vocabulary, p is a property and the pair (s, o) belongs to the extension of the property p. An interpretation assigns the value true to an RDF graph if it assigns the value true to all of its triples.

Additionally, in order to assign a truth value for a graph containing blank nodes, an interpretation must rely on a mapping from the set of blank nodes present in the graph to terms in the graph. This mapping of blank nodes ensures that all occurrences of the same blank node are mapped to the same interpretation element and, since this mapping is not an integral part of the interpretation, it also ensures that blank nodes have no visibility outside the graph. Based on Muñoz et al. (2007), we define the notion of *map*:

Definition 2.5 (Map)**.** *A* map *is a function* $\theta : \mathbf{UBL} \to \mathbf{UBL}$ *preserving URIs and literals, i.e.* $\theta(t) = t$*, for all* $t \in \mathbf{UL}$*. Given a graph* G*, we define* $\theta(G) = \{ (\theta(s), \theta(p), \theta(o)) \mid (s, p, o) \in G \}$*. We speak of a map* θ *from* G_1 *to* G_2*, and write* $\theta : G_1 \to G_2$*, if* θ *is such that* $\theta(G_1) \subseteq G_2$*. Furthermore, we say that a map* θ *is a* grounding *of a graph* G*, iff* $\theta(G)$ *is a ground graph.*

We next present the definition of interpretation according to Muñoz et al. (2007):

Definition 2.6 (Interpretation, Muñoz et al. (2007))**.** *An* interpretation $\mathcal{I}$ *over a vocabulary* V *is a tuple* $\mathcal{I} = \langle \Delta_R, \Delta_P, \Delta_C, \Delta_L, P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}} \rangle$*, where* Δ_R, Δ_P*,* Δ_C, Δ_L *are the interpretation domains of* $\mathcal{I}$*, which are finite non-empty sets, and* $P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}}$ *are the interpretation functions of* $\mathcal{I}$*. They have to satisfy:*

1. Δ_R *are the resources (the domain or universe of* $\mathcal{I}$*);*
2. Δ_P *are property names (not necessarily disjoint from* Δ_R*);*
3. $\Delta_C \subseteq \Delta_R$ *are the classes;*
4. $\Delta_L \subseteq \Delta_R$ *are the literal values (containing* $\mathbf{L} \cap V$*);*
5. $P[\![\cdot]\!]$ *is a function* $P[\![\cdot]\!] \colon \Delta_P \to 2^{\Delta_R \times \Delta_R}$*;*
6. $C[\![\cdot]\!]$ *is a function* $C[\![\cdot]\!] \colon \Delta_C \to 2^{\Delta_R}$*;*
7. $\cdot^{\mathcal{I}}$ *maps each* $t \in \mathbf{UL} \cap V$ *into a value* $t^{\mathcal{I}} \in \Delta_R \cup \Delta_P$ *such that* $\cdot^{\mathcal{I}}$ *is the identity for plain literals and assigns an element in* Δ_R *to each element in* $\mathbf{L}$*.*

2.4.3. RDF Schema

As briefly presented in the previous section, RDFS is a vocabulary that allows for the description of relations between RDF resources. For this thesis, we will rely on a fragment of RDFS, called ρdf, presented by Muñoz et al. (2007), that covers essential features of RDFS. ρdf consists of the following subset of the RDFS vocabulary: { `rdfs:subPropertyOf`, `rdfs:subClassOf`, `rdf:type`, `rdfs:domain`, `rdfs:range` }. In the following, for readability purposes, we are using the following abbreviations: sp for `rdfs:subPropertyOf`, sc for `rdfs:subClassOf`, type for `rdf:type`, dom for `rdfs:domain`, and range for `rdfs:range`.

Based on the definition of interpretation (Definition 2.6), we can define the concept of *model* of an RDF graph:

Definition 2.7 (Model (Muñoz et al., 2007))**.** *An interpretation* $\mathcal{I}$ *is a* model *of a ground graph* G*, denoted* $\mathcal{I} \models G$*, if and only if* $\mathcal{I}$ *is an interpretation over the vocabulary* $\rho df \cup universe(G)$ *that satisfies the following conditions:*

Simple:

1. *for each* $(s, p, o) \in G$*,* $p^{\mathcal{I}_A} \in \Delta_P$ *and* $(s^{\mathcal{I}_A}, o^{\mathcal{I}_A}) \in P[\![p^{\mathcal{I}_A}]\!]$*;*

Subproperty:

1. $P[\![\mathsf{sp}^{\mathcal{I}_A}]\!]$ *is transitive over* Δ_P*;*
2. *if* $(p, q) \in P[\![\mathsf{sp}^{\mathcal{I}_A}]\!]$ *then* $p, q \in \Delta_P$ *and* $P[\![p]\!] \subseteq P[\![q]\!]$*;*

Subclass:

1. $P[\![\mathsf{sc}^{\mathcal{I}_A}]\!]$ *is transitive over* Δ_C*;*
2. *if* $(c, d) \in P[\![\mathsf{sc}^{\mathcal{I}_A}]\!]$ *then* $c, d \in \Delta_C$ *and* $C[\![c]\!] \subseteq C[\![d]\!]$*;*

Typing I:

1. $x \in C[\![c]\!]$ *if and only if* $(x, c) \in P[\![\mathsf{type}^{\mathcal{I}_A}]\!]$*;*
2. *if* $(p, c) \in P[\![\mathsf{dom}^{\mathcal{I}_A}]\!]$ *and* $(x, y) \in P[\![p]\!]$ *then* $x \in C[\![c]\!]$*;*
3. *if* $(p, c) \in P[\![\mathsf{range}^{\mathcal{I}_A}]\!]$ *and* $(x, y) \in P[\![p]\!]$ *then* $y \in C[\![c]\!]$*;*

Typing II:

1. *For each* $\mathsf{e} \in \rho df$*,* $\mathsf{e}^{\mathcal{I}_A} \in \Delta_P$
2. *if* $(p, c) \in P[\![\mathsf{dom}^{\mathcal{I}_A}]\!]$ *then* $p \in \Delta_P$ *and* $c \in \Delta_C$

3. *if* $(p, c) \in P[\![\mathsf{range}^{\mathcal{I}_A}]\!]$ *then* $p \in \Delta_P$ *and* $c \in \Delta_C$
4. *if* $(x, c) \in P[\![\mathsf{type}^{\mathcal{I}_A}]\!]$ *then* $c \in \Delta_C$

Entailment among ground graphs G and H behaves as per the model-theoretic semantics: any interpretation that is a model of G is also a model of H. In the case where G and H may contain blank nodes, $G \models H$ if and only if for any grounding G' of G there is a grounding H' of H such that $G' \models H'$.

In Muñoz et al. (2007), the authors define two variants of the semantics: the default one includes reflexivity of $P[\![\mathsf{sp}^{\mathcal{I}}]\!]$ and $C[\![\mathsf{sc}^{\mathcal{I}}]\!]$ over Δ_P and Δ_C, respectively, but herein we are only considering the alternative semantics presented in Muñoz et al. (2007, Definition 4), which omits this requirement. As a consequence, inferences such as $G \models (a, \mathsf{sc}, a)$ are not supported. However, the drawback of this is minimal since such inferences do not add expressive power and are thus of marginal interest.

Deductive System

In what follows, we present the sound and complete deductive system from Muñoz et al. (2007). The system is arranged in groups of rules that capture the semantic conditions of models. In every rule, A, B, C, X, and Y are meta-variables representing elements in **UBL** and D, E represent elements in **UL**. The rules are as follows:

1. Simple:

(a) $\frac{G}{G'}$ for a map $\theta : G' \to G$ (b) $\frac{G}{G'}$ for $G' \subseteq G$

2. Subproperty:

(a) $\frac{(A,\mathsf{sp},B),(B,\mathsf{sp},C)}{(A,\mathsf{sp},C)}$ (b) $\frac{(D,\mathsf{sp},E),(X,D,Y)}{(X,E,Y)}$

3. Subclass:

(a) $\frac{(A,\mathsf{sc},B),(B,\mathsf{sc},C)}{(A,\mathsf{sc},C)}$ (b) $\frac{(A,\mathsf{sc},B),(X,\mathsf{type},A)}{(X,\mathsf{type},B)}$

4. Typing:

(a) $\frac{(D,\mathsf{dom},B),(X,D,Y)}{(X,\mathsf{type},B)}$ (b) $\frac{(D,\mathsf{range},B),(X,D,Y)}{(Y,\mathsf{type},B)}$

5. Implicit Typing:

(a) $\frac{(A,\mathsf{dom},B),(D,\mathsf{sp},A),(X,D,Y)}{(X,\mathsf{type},B)}$ (b) $\frac{(A,\mathsf{range},B),(D,\mathsf{sp},A),(X,D,Y)}{(Y,\mathsf{type},B)}$

The deductive system presented by Muñoz et al. (2007) includes 7 rules, where the missing rules (rules 6-7) handle reflexivity. Furthermore, as noted in Muñoz et al. (2007), the "Implicit Typing" rules are a necessary addition to the rules presented in P. Hayes (2004) for complete RDFS entailment. These represent the case when variable A in (D, sp, A) and (A, dom, B) or (A, range, B), is a property implicitly represented by a blank node.

We denote with $\{ \tau_1, \ldots, \tau_n \} \vdash_{\mathsf{RDFS}} \tau$ that the consequence τ is obtained from the premise $\tau_1, \ldots, \tau_n$ by applying one of the inference rules 2–5 above.

2.5. Comparison of the Data Models

We present a brief comparison of the data models in Table 2.1, focusing on features of the data model and existing query languages.[6] The features of the data model we considered were the logical structure it uses to represent data, whether the model is an intrinsically ordered data model, and if it provides

[6] In this table we are using the term RDB as a shorthand for the relational model.

Table 2.1.: Feature overview of data models

	RDB	XML	JSON	RDF
Model:				
logical structure	relations	tree	tree	graph
ordered	×	√	√/×	×
schema validation	√	√	×	×
inference	×	×	×	√
Languages:				
query	√	√	×	√
data manipulation	√	√	×	×
schema manipulation	√	×	×	×

means, possibly external, of performing schema validation. Inference capabilities allow to deduce new data based on existing one by specifying structural and representational properties. Regarding the languages, we represent the existence of languages for querying data contained in the respective model, as well as manipulation languages for both data and schema. Such languages can be used, for example, for inserting and updating data or changing the representation structure of data.

The Relational Model. As we have discussed in Chapter 1, the relational model is used to store information for many software applications. A relational database consists of a set of relations (also commonly known as tables), and data for each table is called a record. As the data model overview presented in Table 2.1 shows, the relational model is the most mature, having stable query and manipulation languages.

XML. With the uptake of the WWW, more flexible data models were introduced, such as XML. Also Web pages are described using the HTML language, a syntax that although similar to XML is mostly focused on rendering the contents in a *web browser*. XML is more concerned with describing the data it contains while rendering is instead left to the external XSLT transformation language. XML is a tree-based, ordered data representation format that imposes no restrictions on its element and attribute names, nor on the nesting structure. The XML query and transformation languages were presented in Section 3.2, mostly focusing on the XQuery language. A recommendation for an XQuery Update language was also presented by Robie et al. (2011).

JSON. Section 2.3 presented another tree-based representation format, JSON, that has recently gained traction and uptake on the Web due to its easy integration with the JavaScript language, which is supported by all modern web browsers. JSON is mostly regarded as an interchange format, notably lacking the specifications of any type of query language and schema validation. The JSON data model is also tree-based and it distinguishes different structures (*objects* and *arrays*), where objects consist of unordered sets, while arrays represent an ordered sequence of elements.

RDF. The advance of the traditional, human-readable Web into a machine-readable *Semantic Web* (Berners-Lee, Hendler et al., 2001) introduces a new data model: RDF. RDF is a graph-based data model and, as discussed in Section 1.2, is suitable for representing integrated data. One main difference between RDF and the other data models relates to its capabilities for deducing new data, based on a specialised vocabulary called RDF Schema. RDFS, as opposed to XML Schema, does not behave as a form of data validation but rather as a form of deducing new data. Although the new SPARQL 1.1

query language includes the specification of an update language (Gearon et al., 2012), this is still not a finalised W3C standard so we chose to omit it from Table 2.1. Possible forms of validating RDF data, even though no recommendation exists, may involve (i) using the SPARQL query language (presented in Section 3.3) for determining if any triples do not match the constraints; (ii) SPIN (Knublauch et al., 2011) is a vocabulary that also allows to specify constraints for RDF data; or (iii) by using extensions of OWL towards integrity constraints, e.g. Tao et al. (2010). In this thesis we focus primarily on the RDF data model; however, several other graph-based database models exist and a survey is presented by Angles and Gutiérrez (2008a).

2.6. Conclusion

This chapter introduced the basis for the different data models we are considering in this thesis. As such we described the relational, XML, and RDF data models and included a description of the JSON interchange format. As we have discussed in Section 1.2, from a data integration perspective, a flexible format for representing data is desirable, hence the XML or RDF formats are preferred over the relational model. The major differences between these data models are (i) the structure (table vs. tree vs. graph) that is used to represent data; and (ii) the ordering of the data model (XML is an intrinsically ordered data model, JSON included the ordered array structure, while relational databases and RDF consist of an unordered set of statements).

The specific query language for each of these data models are presented in Chapter 3. These different data models are bridged in our novel transformation language, described in detail in Chapter 4. Furthermore, Chapter 6 presents a proposed extension to the RDF model to represent context information, such as temporal or provenance information, a much needed feature when considering integrated data.

3. Query Languages

Query languages allow users to select and transform data from large sources. The ability to select only relevant data is an essential feature to minimise serialisation and communication overheads, especially when we consider the transmission of data over the Web.

Due to the specific characteristics of each data model, query languages are usually tailored to work with a single data model. For the data models presented in the previous section, the respective query languages are SQL, XQuery and SPARQL, for which we will give an overview next. We also present the closely related XSLT transformation language for XML data.

In Table 2.1 we presented a high-level overview of the available languages for each data model. Data and schema manipulation languages are widely available for relational data, for XML an update language has been recently standardised (Robie et al., 2011) while for RDF data this feature is only included in the upcoming version of SPARQL 1.1 (Gearon et al., 2012).

In the following sections we start by presenting a short overview of the possible forms of querying relational databases, including the SQL query language, before turning to the different query languages for XML in Section 3.2. In this section we again present a short overview of the XPath and XSL Transformations (XSLT) languages and then focus in more detail on the XQuery language, which will be the basis for the XSPARQL language in Chapter 4. Finally, Section 3.3 provides a detailed description of the SPARQL query language for RDF (which we will extend in Chapter 6).

3.1. Querying Relational Databases

In this section we give an overview of conjunctive queries which, according to Abiteboul, Hull et al. (1995), represent the vast majority of relational database queries that are relevant in practice. Later we present the SQL query language, which is the most used query language for relational databases.

3.1.1. Conjunctive queries

In line with the different views on relational data (presented in Section 2.1), conjunctive queries can be formalised under different, although equivalent, perspectives: logic programming and the relational algebra. The logic programming approach follows the corresponding view on relational data presented in Section 2.1, while the relational algebra approach relies on the conventional view. We then present the SQL query language and provide an overview of its mapping into relational algebra.

Under the logic programming approach, in addition to the sets of relations $\mathbf{R}$, attributes $\mathbf{A}$, and values $\mathbf{D}$, we rely on the set of variables $\mathbf{V}$ that range over elements of $\mathbf{D}$. We can now extend the notion of fact to *atom*: an atom over a relation $r \in \mathbf{R}$ is an expression $r(e_1, \ldots, e_n)$ where n is the arity of r and each $e_i \in \mathbf{DV}$ is called a *term*. A fact can also be referred to as a *ground atom*. The notion of *query* can then be defined as:

Definition 3.1 (Rule-based conjunctive query (Abiteboul, Hull et al., 1995))**.** *Given a database schema S, a* rule-based conjunctive query *q over S is an expression of the form:*

$$q(u) \leftarrow r_1(u_1), \ldots, r_n(u_n) \tag{3.1}$$

where each $r_i, i \in [1, n]$ is a relation name from S and each $r_i(u_i)$, is an atom over r_i. Any variable occurring in u must be safe*, i.e. it must also occur at least once in any $u_1, \ldots, u_n$. Furthermore, we denote the set of variables present in q as $vars(q)$.*

A rule-based conjunctive query can be referred to simply as a *rule* where the lefthand side of '$\leftarrow$' is called the *head* and the righthand side is called the *body* of the rule. For example, in Rule (3.1), the head is $q(u)$ and the body corresponds to $r_1(u_1), \ldots, r_n(u_n)$. A rule can be interpreted as: the head atom can be deduced if there are values for the variables in the rule that make the body hold. Given a set of variables $V \subset \mathbf{V}$, a *mapping* (or *valuation*) over V is a function $v : V \to \mathbf{D}$. This function can be extended to represent the identity over any element of $\mathbf{D}$ and thus can map any atom with variables to a fact by applying it to all elements of the atom. For any atom t, the mapping of v is denoted as $v(t)$. Based on this notion of mapping, the *answers* to a query can be defined as:

Definition 3.2 (Answers of a query)**.** *Let $q = q(u) \leftarrow r_1, \ldots, r_n$ be a rule-based conjunctive query and I be a database instance. An* answer *of I under q is:*

$$q(I) = \{\, v(r) \mid v \text{ is a mapping over } vars(q) \text{ and } v(r_i) \in I \text{ for each } i \in [1, n] \,\}$$

As we will see in the next section, since duplicate removal is a computational expensive operation, SQL maintains duplicates in the answers of a query (thus represented as a multiset) unless otherwise instructed.

Another paradigm for relational queries is the algebraic paradigm, which is defined by specifying operations on relation instances, called the *relational algebra*. The three primitive algebra operators are the *selection* (σ), *projection* (π) and the *cartesian product* ($\times$) operators. The full set of operators that form the *relational algebra* also include the *join* ($\bowtie$), *union* ($\cup$) and *difference* ($-$) operators. The selection operator consists of restricting the tuples present in a relation according to a specified condition. The projection operator is used to discard attributes of a relation while the cartesian product combines any two relations and produces a new relation that includes all the attributes of both relations. The join operator consists of combining the projection and cartesian product operators: the result of this operator consists of all the tuples from both relations that have a common value on any common attributes. The union ($\cup$) and difference ($-$) algebra operators are defined as the standard set-theoretical operators.

Considering two relational instances I and J (with sorts U and W, respectively), the relation attributes A and B, and a constant $c \in \mathbf{D}$, the relational algebra operators are defined as:

selection: The two forms of the selection operator, $\sigma_{A=c}$ and $\sigma_{A=B}$ select tuples that match a constant or where the value of two attributes is the same, respectively:

$$\sigma_{A=c}(I) = \{\, t \in I \mid t(A) = c \,\}$$
$$\sigma_{A=B}(I) = \{\, t \in I \mid t(A) = t(B) \,\}$$

projection: This operator consists of restricting the attributes present in a relation. Given a set of attributes $X \subseteq U$, the projection operator returns:

$$\pi_X(I) = \{\, t[X] \mid t \in I \,\}$$

join: The join operator between I and J produces a relation with sort $U \cup W$, such that:

$$I \bowtie J = \{\, t \mid sort(t) = U \cup W \text{ and } t[U] = u \text{ and } t[W] = w \text{ for some } u \in I \text{ and } w \in J \,\}$$

3.1.2. SQL

The SQL is the most widely available query language for relational data, supported by most commercial relational database management systems (Abiteboul, Hull et al., 1995) and is an American National

Standards Institute (ANSI) standard. The core of SQL queries consist of the commonly known *select-from-where* queries, which are equivalent in expressivity to conjunctive queries. An example of a SQL query is shown in Example 3.1.

Example 3.1 (SQL query)**.** The following SQL query, when executed against a database instance following the schema presented in Example 2.2, extracts the names of the artists that are in the "`Nightwish`" band:

```
SELECT persons.personName
FROM persons, bands
WHERE persons.bandId = bands.bandId
      AND bands.bandName = 'Nightwish'
```

The `select` keyword specifies the attributes that should be present in the query results, while `from` specifies the relations names over which the query will be evaluated. It is possible to write '`*`' in place of attribute names in a `select` clause if all the attributes in the relations specified in the `from` clause are to be returned. In SQL, relation names are considered as variables that range over tuples occurring in the corresponding relation and, as shown in Example 3.1, can be used in the `where` clause to specify the relations and attributes. If a query requires more than one variable ranging over the same relation, they can be specified in the `from` clause and assigned different *aliases* for the relation, e.g. `FROM person p1, person p2`. Furthermore, if the attributes are distinct, it is possible to omit the relation name from the `select` query. Finally, the `where` keyword specifies the conditions that any result of the query must satisfy in order to be included in the result set and can express conjunction, disjunction, negation, and nesting. It is possible to completely omit the `where` clause and, in this case, all tuples of the cartesian product of the relations specified in the query are returned.

It is also possible to represent nested queries by using the keywords `in` and `not in`. These keywords behave as operators over sets, testing the inclusion (or not) of an element in the set resulting from the nested query.

The result of a SQL *select-from-where* query evaluation consists of a multiset of tuples, i.e. there may be repeated answers in the results. The explicit use of the `distinct` keyword after the `select` keyword removes any duplicate answers from the resulting set. SQL also includes several aggregate functions, such as `count`, `sum`, and `average`, which perform the specified function over the resulting multisets, such as counting number of elements in the collection, or adding the elements of a multiset composed of numeric elements. Furthermore grouping of tuples, by means of the `group by` operator, also allows to create collections of tuples over which aggregates can be applied.

Included in the SQL specification is also the definition of a DML. This part of the SQL language allows the schema of a relational database to be manipulated, for instance creating new relations, altering the structure of existing ones, or removing existing relations.

Translation to Relational Algebra

A translation from a subset of SQL into relational algebra was presented by Ceri and Gottlob (1985), while the semantics catering for the full syntax of SQL and the three-valued logic inherent with nulls, was presented by Negri et al. (1991). In this thesis, we follow the translation of SQL *select-from-where* queries into relational algebra as presented by Abiteboul, Hull et al. (1995): (i) the `select` keyword behaves as the projection operator π; (ii) the `from` keyword corresponds to the cartesian product operator $\times$; and (iii) the `where` keyword specifies a selection operation σ. In the rest of the thesis, we denote this translation of a SQL query S into relational algebra as $RA_{sql}(S)$.

Table 3.1.: Mapping from SQL to XML datatypes

SQL datatype	XML datatype
character string	xs:string
numeric, decimal	xs:decimal
boolean	xs:boolean
smallint, integer, bigint	xsd:integer
float, real, double precision	xsd:double
date	xsd:date
time	xsd:time
timestamp	xsd:dateTime

Example 3.2 (SQL translation into Relational Algebra)**.** The translation into relational algebra of the query in Example 3.1 is:

$$\pi_{persons.personName}(\sigma_{persons.bandId=bands.bandId \wedge bands.bandId='Nightwish'}(persons \times bands))$$

Mapping SQL Results to XML

The mapping from SQL datatypes into XML Schema datatypes is defined in the SQL specification and was presented in Eisenberg and Melton (2001). An overview of this mapping is presented in Table 3.1. Since XML datatypes generically allow a wider range of valid values, it is common for concrete mappings to impose further restrictions on XML datatypes, however, for this thesis we will omit these restrictions on the XML datatypes. Later, in Section 4.2, we will rely on this mapping of datatypes for the translation of SQL to XML and we refer to the XML representation of SQL values as $sql2xml(SQLValue)$ and vice-versa as $xml2sql(XMLValue)$.

3.2. Querying XML

As for querying XML data, there are different alternatives, most notably XSLT and XQuery. Although these languages are very similar and both tackle similar problems, their fundamental difference is that XSLT was designed to perform transformations between different XML formats, mostly considered for styling and display of information on the Web, while XQuery is focused on querying parts of an XML document or tree (Katz et al., 2003). Notably, XSLT uses an XML syntax for specifying the transformations while XQuery provides a non-XML syntax that aims to be familiar for SQL users. Both of these languages rely on a common core, the XML Path Language (XPath), that allows nodes of an XML document to be selected. Although XPath was being designed at the same time as the XSLT language, it was published as a separate standard by the W3C, who envisioned its use in other languages or even as a single standalone language.

In this section, we start by explaining the XPath language, followed by an high-level overview of the XSLT language. Finally we present the XQuery language, that is used as a basis for the syntax and semantics of XSPARQL.

3.2.1. XPath

XPath (Bray, Paoli and Sperberg-McQueen, 2010) consists of a common core language that is reused by both XSLT and XQuery. The main purpose of XPath is to access specific nodes of an XML document, which it does by providing a non-XML syntax for navigating though the structure of an XML document and selecting the relevant nodes. In XPath, an *expression* is the basic construct of the language and may consist, among other elements, of variable references, function calls, or location paths. Such expressions are evaluated with respect to an *expression context*, which contains the necessary information to determine

the output of an expression, most notably the *context node*: the XML element over which the expression will be evaluated. The evaluation of an XPath expression results in an *object* that can consist of a node-set, boolean, number, or a string.

Location paths. Location paths are an especially important form of XPath expressions since they specify how to navigate through the XML document. A location path consists of a sequence of *location steps* that are separated by the '`/`' character. The result of the evaluation of each location step is the set of nodes and each step in the location path is applied to the set of nodes resulting from the previous step. A location step can be composed of three parts: an *axis*, a *node test*, and any number of *predicates*, which are described next.

Axis: The axis is used to select nodes by specifying the relation that the selected nodes should have from the context node. Some of the available axes allow the `child`, `parent`, `following-sibling`, or `preceding-sibling` of the context node to be selected. These axes correspond to selecting, relative to the context node, all the nodes that are one level below (`child`), one level up (`parent`), at the same level after (`following-sibling`) or before (`preceding-sibling`) the context node. Furthermore, the `attribute` axis can be used to select all the attributes of the context node, "`self`" refers to the context node and "`descendant-or-self`" to the context node and its descendants.

Node Tests: Node tests can be combined with axes to further restrict the selected set of nodes. For example "`child::band`" will select all the children of the context node with "`band`" name. Similarly, '`*`' can be used to select all elements: "`attribute::*`" selects all the attributes of the context node.

Predicates: Predicates can be used to further filter a node-set and produce a new node-set: given a node-set, the predicate expression is evaluated by using each node in the node set as the new context node. If the evaluation of the predicate yields true, then the node is included in the newly created node set.

The XPath specification also defines an abbreviated syntax for representing location steps, where the expression can be specified by either assuming a default axis or by specifying shortcuts. For example, the "`child`" axis is the default and can thus be omitted. The '`@`' abbreviation can be used for selecting the *attribute* axis, thus, "`@name="Nightwish"`" is short for "`attribute::name="Nightwish"`". Other available abbreviations are '`//`' for "`/descendant-or-self::node()/`", '`.`' for "`self::node()`" and '`..`' for "`parent::node()`".

Example 3.3 (XPath expression)**.** The following XPath expression:

```
//band[@name="Nightwish"]/members
```

which is an abbreviated form of the expression:

```
/descendant-or-self::node()/child::band[attribute::name="Nightwish"]/child::members
```

when executed over the XML document presented in Data 2.1, returns the "`band`" XML element whose value of the "`name`" attribute is "`Nightwish`".

Further expressions available in the XPath language include `for` expressions that, in a somewhat similar fashion to imperative programming languages,[1] allow to repeat an expression (called *return expression*) for different values of the *range variable* (detailed in Section 3.2.3), conditional expressions and quantified expressions. Conditional expressions allow to execute different expressions (*then expression* or *else expression*) depending on the result of the *if expression* and quantified expressions can be used to test whether an expression is true for all (`every`) or at least one (`some`) members of a sequence.

[1] We say somewhat similar since the evaluation order of the return expression is not imposed by XPath.

3.2.2. XSLT

XSL Transformations (XSLT) (Kay, 2007) is a transformation language for XML that allows to manipulate XML documents by matching subsets of the XML structure and specifying transformation rules for the matched elements. The syntax of XSLT is XML-based and defines a set of XML elements that are interpreted as XSLT instructions, distinguished by using the namespace "`http://www.w3.org/1999/XSL/Transform`", commonly abbreviated by the `xsl`: prefix. As previously noted, XSLT relies on the XPath language to navigate and access the elements in the XML document.

XSLT transformations are called *stylesheets*, referring to the origins of the language, mostly used for defining the style of an XML or XHTML document for presentation in a web browser. XSLT follows the functional programming paradigm, where the stylesheet defines a set of rules that produce the output tree as a function of the input tree. Such rules, called *template rules*, are applied to the source or input tree in order to produce the result (or output) tree. The part of the rule that is matched against the XML elements in the source is called the *pattern*, while the *sequence constructor* part is instantiated by elements matched from the source tree in order to produce the result tree. Recalling the logic programming approach for querying relational databases (Section 3.1), the pattern can be considered the body of the query, while the sequence constructor can be considered the head.

Template rules are defined using XML elements named `xsl:template` (as presented in Example 3.4). The "`match`" attribute is used to specify the XML elements to which the template will be applied, while the body of the template defines the output. The recursive application of template rules is selected by the "`xsl:apply-templates`" element, possibly specifying which XML elements from the input should be matched by providing an XPath expression as the value of the "`select`" attribute. Whenever this attribute is omitted, the default is to select all children of the context node.

Example 3.4 (XSLT template rules)**.** The following template rule selects the band element whose value for the name attribute is "`Nightwish`":

```
<xsl:template match="bands">
    <xsl:apply-templates select="//band[@name='Nightwish']/members"/>
</xsl:template>
```

While the next template rule simply outputs all members of the selected bands:

```
<xsl:template match="member">
  <xsl:apply-templates/>
</xsl:template>
```

Combining these two template rules in an XSLT stylesheet will make the stylesheet output the names of members of the "`Nightwish`" band, when applied to Data 2.1.

Similar to XPath, an XSLT stylesheet is evaluated with regards to an *expression context* and relies on the XPath specification for defining the contents of each expression context. Each template rule is evaluated by specifying the matched input XML element as the context node.

The XSLT specification further defines instructions for specifying repetition (`xsl:for-each`), conditional processing (`xsl:if` and `xsl:choose`), variable declaration (`xsl:variable`), and function declaration (`xsl:function`).

3.2.3. XQuery

XQuery (Chamberlin et al., 2010) has been the W3C recommended query language for XML since early 2007. There are several similarities between XQuery and XSLT and both query languages can address

the same use cases. Some of the most evident similarities include: (i) declarative semantics supporting single-assignment variables; (ii) the use of XPath for selecting input XML elements; (iii) the construction of new XML elements explicitly and at runtime; and (iv) support for user-defined functions.

As we have seen, XSLT was designed as a transformation language for XML documents, focusing on transformations that facilitate displaying data for the user. On the other hand, XQuery behaves more like a query language, aiming at extracting data from collections or large individual XML documents. These design choices are apparent even in the syntax of the languages, where XQuery follows a non-XML syntax. However, XQuery reuses other XML-based specifications such as the XDM data model and XSD. Any input document for an XQuery query, commonly specified using the `fn:doc` function, is translated into an XDM instance and the respective query is executed over this abstract structure.

As in XPath and XSLT, also in XQuery the basic construct of a query is called an *expression*. In XQuery, expressions are mostly composed of FLWOR expressions. This name stems from the available expressions: `for`, `let`, `where`, `order by`, and `return`.

Definition 3.3 (Tuple Stream)**.** *In a FLWOR expression, the result of the evaluation of "`for` $v" and "`let` $v" clauses consists of an ordered sequence of elements that $v is bound to. Following the XQuery specification, we refer to this sequence as* tuple stream*.*

Optionally, this produced sequence can be filtered using the `where` clause or ordered with the `order by` clause. Finally, the `return` clause is evaluated for every element of the resulting sequence and each result is included in the sequence produced by the FLWOR expression. Any XQuery variable is represented by using an expanded QName also making it possible to disambiguate variables based on declared prefixes. Further details are available in Draper, Fankhauser et al. (2010, Section 3.1.1.1). Another important feature is that `for` clauses may optionally include a positional variable in the form of "`for` $*Var* `at` $*PosVar*". In this case, for each evaluation of the `return` expression, $*PosVar* is assigned an integer corresponding to the position of $*Var* in the tuple stream.

Definition 3.4 (Expression Context)**.** *Similar to XPath and XSLT, any XQuery expression E is evaluated with regards to an* Expression Context *that holds the static environment (*statEnv*) and the dynamic environment (dynEnv) up until the evaluation of E.*

Environments include different components that hold the necessary information for the evaluation of any XQuery expression: statEnv holds the information available during static analysis, for example the varType component holds variable type information. The dynEnv environment contains information available during expression evaluation, like the value for variables, stored in the varValue component. Given an expression context C, we refer to the static environment of C as statEnv(C) and to the dynamic environment as dynEnv(C). Different components can be accessed via their name: statEnv(C) .varType and the specific value of the environment element *var* can be accessed using statEnv(C) .varType(*var*). If the expression context C is not explicitly presented, statEnv and dynEnv can be used in place of statEnv(C) and dynEnv(C).

Example 3.5 (XQuery query)**.** The slightly verbose XQuery equivalent to the XSLT presented in Example 3.4 is the following query:

```
for $member in //band[@name='Nightwish']//member
let $memberName := $member/text()
return $memberName
```

Executing this query over the XML document presented in Data 2.1, again returns the members of the "`band`" XML element whose value of the "`name`" attribute is "`Nightwish`".

XQuery allows to write arbitrary queries, it is actually a Turing complete language (Kepser, 2004), for instance the `return` part of a FLWOR expression may contain other (nested) FLWOR expressions. In such cases we commonly refer to the first FLWOR expression as the *outer* query, while the FLWOR expression that is containted inside the `return` is refered to as the *inner* query.

Semantics

The semantics of XQuery (Draper, Fankhauser et al., 2010) is defined in terms of (i) *normalisation rules*, (ii) *static typing rules*, and (iii) *dynamic evaluation rules*. Normalisation rules reduce the syntax of XQuery to an abstract syntax denoted XQuery Core: a subset of XQuery that, while semantically equivalent, aims to be easier to define, implement and optimise (Katz et al., 2003). Static typing rules are applied over the XQuery Core language and are used to assign a type to each XQuery expression. The dynamic evaluation rules are responsible for producing the results of each expression while guaranteeing that its input is consistent with the previously determined typing information.

In this thesis we will use the term *bound variable* to refer to a variable that has been previously declared in an query, for example, $\$v$ is considered bound if it has been previously declared by a "`for` $\$v$" or "`let` $\$v$" expression.

The complete semantics of XQuery is defined by specifying *normalisation*, *static* and *dynamic evaluation* rules for each expression of the language and, as an example, we next present the rules of the XQuery `for` expression.

Normalisation Rules. *Normalisation rules* are represented using mapping rules, where $[\![\cdot]\!]_{Expr}$ represents the XQuery expression to be matched, while the resulting XQuery Core expression is included after the == separator. Furthermore, fixed-width font (like `for` and `in`) refer to specific keywords, and *italic* font refers to productions in the XQuery Core grammar (Draper, Fankhauser et al., 2010, Appendix A). The following example shows the application of the normalisation rules over consecutive *ForClause*s – considered a shorthand syntax – into nested *ForClause*s in XQuery Core:

$$
\begin{aligned}
&\left[\!\!\left[\begin{array}{l}
\texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ \mathit{Expr}_1 \\
\quad ,\cdots, \\
\quad \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ \mathit{Expr}_n \\
\quad \mathit{ReturnClause}
\end{array}\right]\!\!\right]_{Expr} \\
&\qquad\qquad == \qquad\qquad (\text{N1}) \\
&\texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ [\![\mathit{Expr}_1]\!]_{Expr}\ \texttt{return} \\
&\quad \cdots \\
&\quad \texttt{for}\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ [\![\mathit{Expr}_n]\!]_{Expr} \\
&\quad [\![\mathit{ReturnClause}]\!]_{Expr}
\end{aligned}
$$

The normalisation process consists of the recursive application of the defined rules over each expression in the language.

Static and Dynamic Evaluation Rules. On the other hand, *static type rules* and *dynamic evaluation rules* are represented using inference rules of the form:

$$\frac{\mathit{premise}_1 \ \cdots \ \mathit{premise}_n}{\mathit{conclusion}}$$

Rule premises are composed of the so-called *judgements* and such judgements are said to *hold* if they are considered true. Some judgments used in this thesis are:

Type. The judgment:

$$\text{statEnv} \vdash \mathit{Expr} : \mathit{Type}$$

holds if, in the static environment statEnv, the expression *Expr* has the type *Type*. Also related to typing, the *prime* and *quantifier* are functions that extract all the item types of its parameter and try to estimate the number of items in a type (?, +, or *), respectively.

Variable Expansion. Similarly,

$$\text{statEnv} \vdash \textit{VarName} \textbf{ of var expands to } \textit{Variable}$$

holds if *Variable* corresponds to the expanded QName of *VarName*.

Context Extension. Contexts can be extended by using the '+' notation, for example:

$$\text{statEnv} + \text{varType}(\textit{Variable}_{pos} \Rightarrow \texttt{xs:integer})$$

creates a new context based on statEnv by adding the information that $\textit{Variable}_{pos}$ is of type `xs:integer` to the varType component of statEnv.

Expression Evaluation. A commonly used judgment in dynamic evaluation rules is

$$\text{dynEnv} \vdash \textit{Expr} \Rightarrow \textit{Value}$$

which holds if, in the environment dynEnv, the expression *Expr* evaluates to the value *Value*.

As an example of the use of these judgments, the following static type rule handles the typing of a `for` clause with a positional variable:

$$\frac{\begin{array}{c}\text{statEnv} \vdash \textit{Expr}_1 \colon \textit{Type}_1 \\ \text{statEnv} \vdash \textit{VarName} \textbf{ of var expands to } \textit{Variable} \\ \text{statEnv} \vdash \textit{VarName}_{pos} \textbf{ of var expands to } \textit{Variable}_{pos} \\ \text{statEnv} + \text{varType}\left(\begin{array}{l}\textit{Variable} \Rightarrow \textit{prime}(\textit{Type}_1)\,; \\ \textit{Variable}_{pos} \Rightarrow \texttt{xs:integer}\end{array}\right) \vdash \textit{Expr}_2 \colon \textit{Type}_2\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$\textit{VarName} \texttt{ at } \textit{VarName}_{pos} \texttt{ in } \textit{Expr}_1 \\ \texttt{return } \textit{Expr}_2 \colon \textit{Type}_2 \cdot \textit{quantifier}(\textit{Type}_1)\end{array}} \tag{S1}$$

The dynamic evaluation of `for` expressions consists of first evaluating the expression specified by the `in` clause and, for each element of the resulting sequence, assign it to the `for` variable and then evaluating the `return` expression. Hence, the semantics separates the dynamic evaluation rules of `for` expressions into two cases, depending on whether the `in` expression returns any elements. If the `in` expression evaluates to an empty sequence, the `for` expression also evaluates to an empty sequence:

$$\frac{\text{dynEnv} \vdash \textit{Expr}_1 \Rightarrow ()}{\text{dynEnv} \vdash \texttt{for } \$\textit{VarName}\ \textit{OptPositionalVar} \texttt{ in } \textit{Expr}_1 \texttt{ return } \textit{Expr}_2 \Rightarrow ()} \tag{D1}$$

Otherwise, the dynamic evaluation rule of a `for` clause with a positional variable is presented next:

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \textit{Expr}_1 \Rightarrow \textit{Item}_1, \cdots, \textit{Item}_n \\ \text{statEnv} \vdash \textit{VarName} \textbf{ of var expands to } \textit{Variable} \\ \text{statEnv} \vdash \textit{VarName}_{pos} \textbf{ of var expands to } \textit{Variable}_{pos} \\ \text{dynEnv} + \text{varValue}\left(\begin{array}{l}\textit{Variable} \Rightarrow \textit{Item}_1; \\ \textit{Variable}_{pos} \Rightarrow 1\end{array}\right) \vdash \textit{Expr}_2 \Rightarrow \textit{Value}_1 \\ \vdots \\ \text{dynEnv} + \text{varValue}\left(\begin{array}{l}\textit{Variable} \Rightarrow \textit{Item}_n; \\ \textit{Variable}_{pos} \Rightarrow n\end{array}\right) \vdash \textit{Expr}_2 \Rightarrow \textit{Value}_n\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\textit{VarName} \texttt{ at } \textit{VarName}_{pos} \texttt{ in } \textit{Expr}_1 \\ \texttt{return } \textit{Expr}_2 \Rightarrow \textit{Value}_1, \cdots, \textit{Value}_n\end{array}} \tag{D2}$$

Another judgement used for matching element values to types is "statEnv $\vdash$ *Value* `matches` *Type*", which holds when, in the static environment statEnv, the type of *Value* is *Type* or can be derived from *Type* (as presented in Section 2.2.2).

3.3. Querying RDF with SPARQL

This section provides an overview of the SPARQL query language, which is the W3C recommended query language for RDF. We present the syntax and semantics of SPARQL and wrap-up with an overview of the new features introduced by the forthcoming update to the SPARQL language, dubbed SPARQL 1.1. The W3C SPARQL specification consists of the following documents:

(i) a *query language* for RDF (Prud'hommeaux and Seaborne, 2008);
(ii) a protocol describing the interactions between a *query engine* and *query clients* (Clark et al., 2008); and
(iii) the XML serialisation of the results of a `select` and `ask` query (Beckett and Broekstra, 2008).

We will focus on the description of the SPARQL query language for RDF by following the W3C specification and the semantics presented by Pérez et al. (2009).

Syntax

A *SPARQL query* is defined by a triple $Q = (P, G, V)$, where P is a *graph pattern*, G is an RDF *dataset* and V is the *result form*. Considering a setting similar to rule-based query answering for relational databases, a SPARQL query can also be viewed as: $V \leftarrow P$, where V can be assumed as the *head* of the query, while P is the *body* (Pérez et al., 2009). The next sections describe each component of SPARQL queries, namely RDF datasets, the result form, and graph patterns.

RDF Dataset. An RDF *dataset* forms the input data provided to a SPARQL query and is composed of: (i) exactly one (unnamed) graph considered to be the *default* graph; and (ii) a set of named graphs of the form $\langle n_i, g_i \rangle$, where n_i is a URI corresponding to the *name* of the graph and g_i is an RDF graph. In a SPARQL query, the default graph is specified using `from` clauses, while the named graphs are indicated using `from named` clauses. Since a SPARQL query may contain several `from` clauses, the default graph is taken as the *RDF merge* of graphs specified in all `from` clauses (cf. Definition 2.4).

The notion of *active graph* is introduced in the evaluation semantics of SPARQL to distinguish which RDF graph the basic graph pattern is matched against. At the start of a SPARQL query evaluation, the active graph is the default graph and it is changed when a `graph` keyword is encountered in the graph pattern (as further explained below).

Result Form. The *result form* specifies the output of a SPARQL query and may be one of the following four types:

`select`: returns the matched values (substitutions) for variables present in the query;
`construct`: returns an RDF graph that is created based on the specified *template* and the substitutions obtained by executing the query;
`ask`: returns a boolean indicating if the graph pattern matches any of the data; and
`describe`: returns an RDF graph that contains information regarding the resources contained in the query.

For this thesis we focus primarily on `select` and `construct` queries.[2] In the case of `select` queries, the result form is a set of variables and the result of the query consists of sequences of variable bindings for these variables, determined according to the specified graph pattern. In a `construct` query, as presented in Prud'hommeaux and Seaborne (2008, Section 10.2), the solutions of the graph pattern are used to instantiate the *template* provided. The result of a `construct` query is an RDF graph obtained from the

[2] In Chapter 4 we will refer to these result forms as *SparqlForClause* and *ConstructClause*, respectively.

union of all instantiations of variables in the template that result in valid RDF triples. When a `construct` template contains blank nodes, a different blank node label will be generated for each instantiation of the template, i.e. blank nodes are only shared within the same solution.

Graph Patterns. SPARQL is a graph-matching query language and its syntax directly reflects this. The body (graph pattern) of a SPARQL query consists primarily of *triple patterns* that are matched against the RDF data. Triple patterns are RDF triples, possibly containing variables appearing in subject, predicate or object positions. In the SPARQL syntax, a graph pattern follows the `where` keyword.

A simple form of graph pattern is a set of triple patterns, also called a Basic Graph Pattern (BGP). Here, we present the syntax of SPARQL based on the definitions provided by Pérez et al. (2009), which describes a normalised syntax based on 3-tuples:

Definition 3.5 (Graph Patterns)**.** *Let* $\mathbf{U}$*,* $\mathbf{B}$*,* $\mathbf{L}$ *be defined as before. Furthermore, let* $\mathbf{V}$ *denote a set of variables disjoint from* $\mathbf{UBL}$*,* graph patterns *are inductively defined as follows:*

- *a tuple* $(s, p, o) \in \mathbf{ULV} \times \mathbf{UV} \times \mathbf{ULV}$*, called a* triple pattern*, is a graph pattern;*
- *a set of triple patterns, called a* Basic Graph Pattern (BGP)*, is a graph pattern;*
- *if* P *and* P' *are graph patterns, then* $(P$ `and` $P')$*,* $(P$ `optional` $P')$*, and* $(P$ `union` $P')$ *are graph patterns;*
- *if* P *is a graph pattern and* $i \in \mathbf{UV}$*, then* (`graph` i P) *is a graph pattern; and*
- *if* P *is a graph pattern and* R *is a* `filter` *expression, then* $(P$ `filter` $R)$ *is a graph pattern.*

For any pattern P*, we write* $vars(P)$ *for the set of all variables occurring in* P*. A* `filter` *expression* R *can be composed from constants, elements of* $\mathbf{ULV}$*, comparison operators (‘*$=$*’, ‘*$<$*’, ‘*$>$*’, ‘*$\leqslant$*’, ‘*$\geqslant$*’), logical connectives (‘*$\neg$*’, ‘*$\wedge$*’, ‘*$\vee$*’) and* built-in *functions. Some of the available built-in functions include the unary functions:* `BOUND`*,* `isIRI`*,* `isURI`*,* `isBLANK`*,* `isLITERAL`*,* `STR`*,* `LANG`*, and* `DATATYPE`*. A complete list of built-in functions is included in Prud'hommeaux and Seaborne (2008, Section 11).*

As is common practice in the definition of SPARQL queries, we do not consider blank nodes in graph patterns, and thus do not include them in our definitions. However, this restriction does not affect the expressivity of SPARQL, since blank nodes in query patterns can always be replaced equivalently with variables (Pérez et al., 2009). Although in definitions we rely on an algebraic formalism for the syntax of SPARQL, as per Pérez et al. (2009), in the examples we follow the W3C specification, which can be naturally mapped to the algebraic form, where the `and` operator is represented by a dot (‘`.`’). The mapping between the W3C SPARQL syntax and the algebraic form we use is presented by Arenas, Gutiérrez et al. (2009). Thus, Example 3.6 presents a SPARQL query where the `prefix` keyword declares a URI prefix that is used later in the query.

Example 3.6 (SPARQL query)**.** The following SPARQL query retrieves the names of persons that are members of the “`Nightwish`” band:

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix mo: <http://purl.org/ontology/mo/>

SELECT $personName
WHERE { $band a mo:MusicGroup;
             foaf:name "Nightwish";
             foaf:member $person .
        $person foaf:name $personName }
```

Solution Modifiers. The evaluation of graph patterns generates a sequence of results initially with no specific *order* (further detailed in the following section). Solution modifiers, such as `order by`, `limit`, `offset`, and `distinct` can be applied to this solution sequence. The `order by` modifier is used to specify an ordering for the sequence, specified as a list of variables present in the solution sequence and the direction of the ordering (`ASC` or `DESC`). Furthermore, the `distinct` modifier eliminates any duplicate solutions, while `limit` and `offset` are used to restrict the number of solutions that are returned and to discard solutions from the beginning of the sequence, respectively.

Semantics

The semantics of SPARQL is defined based on the evaluation of BGPs, namely the *matching* of the BGPs against the supplied RDF graph and the algebra that is built on top of this BGP matching. We start by presenting the notion of *solution mappings*, which will be the results of the evaluation of BGPs and then present how *compatible* solution mappings can be combined in order to define the evaluation semantics of SPARQL. This evaluation algebra was presented by Cyganiak (2005); Pérez et al. (2006) and later adapted to the W3C specification (Prud'hommeaux and Seaborne, 2008, Section 12.5).

The matching of BGPs is performed against the previously mentioned *active graph*, a specific RDF graph contained in the dataset of the query. The active graph is initially set to the default graph of the dataset and is changed whenever a `graph` keyword is processed. This matching is represented by a function that maps query variables to RDF terms present in the active graph and is called a *solution mapping*:

Definition 3.6 (Solution Mapping (Prud'hommeaux and Seaborne, 2008))**.** *A* solution mapping *is a partial function mapping SPARQL variables to RDF terms. The* domain *of a solution mapping μ, $dom(\mu)$, is the set of variables for which μ is defined. We denote the value of variable $v \in \mathbf{V}$ according to solution μ as $\mu(v)$.*

The replacement of variables included in a graph pattern according to a solution mapping is defined next.

Definition 3.7 (Variable Substitution)**.** *Let P be a graph pattern and μ be a solution mapping. The* variable substitution of P by μ, *denoted $\mu(P)$, is the graph pattern P with all variables $v \in vars(P) \cap dom(\mu)$ substituted by $\mu(v)$.*

It is worthy to note that if a solution mapping μ contains bindings for all variables in a graph pattern P, i.e. $dom(\mu) = vars(P)$, and all triples in $\mu(P)$ are valid RDF triples, then $\mu(P)$ can be considered an RDF graph. If μ provides bindings only for a subset of the variables present in the graph pattern P, $\mu(P)$ yields another (more restrictive) graph pattern. For the specification of the SPARQL algebra below, we introduce the notion of compatible solution mappings.

Definition 3.8 (Compatible Mappings)**.** *Let μ_1 and μ_2 be solution mappings, μ_1 and μ_2 are* compatible *if and only if for any $v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. The* union *of two compatible mappings μ_1 and μ_2 consists of the standard set-theoretical union $\mu_1 \cup \mu_2$.*

The SPARQL relational algebra (see Cyganiak (2005); Prud'hommeaux and Seaborne (2008); Pérez et al. (2009)) defines how to combine solution mappings. Our semantics of SPARQL is based on the semantics presented by Arenas, Gutiérrez et al. (2009), where the SPARQL algebra operators are extended to the multiset case by preserving the cardinality of solutions:[3]

Definition 3.9 (SPARQL Relational Algebra)**.** *Let Ω_1 and Ω_2 be multisets of solution mappings:*

[3]Following the notation of the operators presented by Arenas, Gutiérrez et al. (2009) we use the standard set operators.

$\Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ compatible} \}$
$\Omega_1 \cup \Omega_2 = \{ \mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2 \}$
$\Omega_1 - \Omega_2 = \{ \mu_1 \in \Omega_1 \mid \text{for all } \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ not compatible} \}$
$\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2)$

The definition of BGP matching from Prud'hommeaux and Seaborne (2008, Section 12.3) specifies the solutions to a query. We denote the evaluation of a BGP P over a graph G as $[\![P]\!]_G$:

Definition 3.10 (Basic Graph Pattern Matching (Prud'hommeaux and Seaborne, 2008, Section 12.3.1))**.** *Given a graph G and a BGP P, a* solution *μ for P over G is a mapping over $V \subseteq vars(P)$ such that $G \models \mu(P)$. Following the definitions presented in Section 2.4, $G \models \mu(P)$, means that any triple in $\mu(P)$ is entailed by G.*

This definition of BGP matching relies on the underlying entailment notion, which according to the SPARQL specification corresponds to simple graph entailment (P. Hayes, 2004). Furthermore, in order to ensure the local scope of blank nodes, query solutions are taken from the *scoping graph*, a graph that is equivalent to the active graph but does not share any blank nodes with it or any graph pattern within the query.

The evaluation semantics of more complex patterns including `filters`, `optional` patterns, `and` patterns, `union` patterns is built on top of this *basic graph pattern matching*, where each SPARQL operator is mapped to an algebra expression:

Definition 3.11 (Evaluation (Pérez et al., 2009, Definition 2.2))**.** *Let $\tau = (s, p, o)$ be a triple pattern, P, P_1, P_2 graph patterns and G an RDF graph, then the evaluation $[\![\cdot]\!]_G$ is recursively defined as follows:*

$[\![\tau]\!]_G = \{ \mu \mid dom(\mu) = var(P) \text{ and } G \models \mu(\tau) \}$
$[\![P_1 \texttt{ and } P_2]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
$[\![P_1 \texttt{ union } P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
$[\![P_1 \texttt{ optional } P_2]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$
$[\![P \texttt{ filter } R]\!]_G = \{ \mu \in [\![P]\!]_G \mid R\mu \text{ is true} \}$

where R is a `filter`[4] expression, $u, v \in \mathbf{UBLV}$. The valuation of R on a substitution μ, written $R\mu$, is true *if:*

(1) $R = \texttt{BOUND}(v)$ with $v \in dom(\mu)$;
(2) $R = \texttt{isBLANK}(v)$ with $v \in dom(\mu)$ and $\mu(v) \in \mathbf{B}$;
(3) $R = \texttt{isIRI}(v)$ with $v \in dom(\mu)$ and $\mu(v) \in \mathbf{U}$;
(4) $R = \texttt{isLITERAL}(v)$ with $v \in dom(\mu)$ and $\mu(v) \in \mathbf{L}$;
(5) $R = (u = v)$ with $u, v \in dom(\mu) \cup \mathbf{UBL} \wedge \mu(u) = \mu(v)$;
(6) $R = (\neg R_1)$ with $R_1\mu$ is false;
(7) $R = (R_1 \vee R_2)$ with $R_1\mu$ is true or $R_2\mu$ is true;
(8) $R = (R_1 \wedge R_2)$ with $R_1\mu$ is true and $R_2\mu$ is true.

$R\mu$ yields an error (denoted ε), if:

(1) $R = \texttt{isBLANK}(v)$, $R = \texttt{isIRI}(v)$, or $R = \texttt{isLITERAL}(v)$ and $v \notin dom(\mu)$;
(2) $R = (u = v)$ with $u \notin dom(\mu) \cup T$ or $v \notin dom(\mu)$;
(3) $R = (\neg R_1)$ and $R_1\mu = \varepsilon$;
(4) $R = (R_1 \vee R_2)$ and $(R_1\mu \neq \text{true}$ and $R_2\mu \neq \text{true})$ and $(R_1\mu = \varepsilon$ or $R_2\mu = \varepsilon)$;

[4] For simplicity, we will omit from the presentation `filters` such as comparison operators ('<', '>', '⩽', '⩾'), data type conversion and string functions. Further details are presented in Prud'hommeaux and Seaborne (2008, Section 11.3).

(5) $R = (R1 \wedge R2)$ *and* $R_1\mu = \varepsilon$ *or* $R_2\mu = \varepsilon$.

Otherwise $R\mu$ *is* false.

The presented definition considers only safe `filters` where, for a pattern "P `filter` R", the filter R is said to be *safe* if $vars(R) \subseteq vars(P)$. However, the SPARQL specification defines that in `optionals`, any filter is scoped to the Group Graph Pattern that contains the `optional`. As such, we include the definition that caters for unsafe `filters`, introduced by Angles and Gutiérrez (2008b):

Definition 3.12 (`optional` with `filter` evaluation)**.** *Let* P_1, P_2 *be graph patterns and* R *a* `filter` *expression. A mapping* μ *is in* $[\![P_1$ `optional` $(P_2$ `filter` $R)]\!]_G$ *if and only if:*

- $\mu = \mu_1 \cup \mu_2$, *s.t.* $\mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$ *are compatible and* $R\mu$ *is true, or*
- $\mu \in [\![P_1]\!]_G$ *and* $\forall \mu_2 \in [\![P_2]\!]_G$, μ *and* μ_2 *are not compatible, or*
- $\mu \in [\![P_1]\!]_G$ *and* $\forall \mu_2 \in [\![P_2]\!]_G$ *s.t.* μ *and* μ_2 *are compatible, and* $R\mu_3$ *is false for* $\mu_3 = \mu \cup \mu_2$.

Finally, the evaluation semantics of SPARQL consists of computing a *sequence of solution mappings*, where any existing solution modifiers are applied to the multiset of results. If no solution modifiers are specified a default ordering is assumed.

Definition 3.13 (Solution sequences)**.** *Sequences of solution mappings are simply referred to as* solution sequences*, often denoted by* Ω.

These conditions of SPARQL `construct` queries, informally specified in Section 3.3, are reflected in the following definition. Later, we will rely on this definition to show the equivalence of the newly introduced XSPARQL `construct` expressions and SPARQL `construct` expressions.

Definition 3.14 (SPARQL `construct` semantics)**.** *Let* C *be a ConstructTemplate and* Ω *a* solution sequence*. The SPARQL* `construct` *returns an RDF graph generated by the set-theoretical union of the triples obtained from substituting variables in* C *with their bindings from* Ω *and satisfying the following conditions:*

(1) any invalid RDF triples that may be produced by the instantiation of the ConstructTemplate are ignored; and
(2) blank node labels within the ConstructTemplate are considered scoped to the template for each solution, i.e. if the same label occurs twice in a template, then there will be one blank node created for each solution in Ω*, but there will be different blank nodes for triples generated by different query solutions. Blank nodes in the graph template be shared only within the same query solution* $\mu_i \in \Omega$.

Query Answering

The SPARQL query language presented in the previous section can be viewed in a similar setting to the rule based conjunctive queries presented for relational databases in Section 3.1. Also inspired by Gutiérrez, Hurtado and Mendelzon (2004), we assume that an RDF graph G is *ground*, where all blank nodes have been skolemised, i.e. consistently replaced with terms in **UL**. A *query* is of the rule-like form:

$$q(\bar{\mathbf{x}}) \leftarrow \exists \bar{\mathbf{y}}.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{y}})$$

where $q(\bar{\mathbf{x}})$ is the *head* and $\exists \bar{\mathbf{y}}.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{y}})$ is the *body* of the query. The body of the query is a conjunction of triples τ_i $(1 \leqslant i \leqslant n)$ and, similar to Section 3.1, we use the symbol ',' to denote conjunction in the rule body. The vectors $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are vectors of variables occurring in the body of the rule called the *distinguished variables* and *non-distinguished variables*, respectively. The variables in $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are disjoint and each variable occurring in τ_i must be either distinguished or non-distinguished.

In a query, we allow *built-in triples* of the form (s, p, o), where p is a *built-in predicate* taken from a reserved vocabulary and having a *fixed interpretation*. We generalise the built-ins to any n-ary predicate p, where p's arguments may be variables from $\mathbf{V}$ and values from $\mathbf{UL}$. We will assume that the evaluation of the predicate can be decided in finite time. For convenience, we write functional predicates[5] as *assignments* of the form $x{:=}f(\bar{\mathbf{z}})$ and assume that the function $f(\bar{\mathbf{z}})$ is safe (according to Definition 3.1). We also assume that a non functional built-in predicate $p(\bar{\mathbf{z}})$ should be safe as well.

Example 3.7 (RDF conjunctive query)**.** An example query is:

$$q(n) \leftarrow (x, \texttt{ex:memberOf}, y), (x, \texttt{foaf:name}, n), (y, \texttt{type}, \texttt{mo:Band}), (y, \texttt{foaf:Name}, \text{“}\texttt{Nightwish}\text{”})$$

which intends to retrieve all persons names n that are members of a band y with the name “`Nightwish`”.

In order to define an *answer* to a query we introduce the following:

Definition 3.15 (Query instantiation)**.** *Given a vector $\bar{\mathbf{x}} = \langle x_1, \ldots, x_k \rangle$ of variables, a* substitution *over $\bar{\mathbf{x}}$ is a vector of terms $\bar{\mathbf{t}}$ replacing variables in $\bar{\mathbf{x}}$ with terms of* **UBL**. *Then, given a query $q(\bar{\mathbf{x}}) \leftarrow \exists\bar{\mathbf{y}}.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{y}})$, and two substitutions $\bar{\mathbf{t}}, \bar{\mathbf{t}}'$ over $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$, respectively, the* query instantiation *$\varphi(\bar{\mathbf{t}}, \bar{\mathbf{t}}')$ is derived from $\varphi(\bar{\mathbf{x}}, \bar{\mathbf{y}})$ by replacing $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ with $\bar{\mathbf{t}}$ and $\bar{\mathbf{t}}'$, respectively.*

Note that, similar to the variable substitution of a solution mapping in SPARQL (cf. Definition 3.7), if all tripes in a query instantiation are valid RDF triples, the query instantiation can be considered an RDF graph.

Definition 3.16 (Entailment)**.** *Given a graph G, a query $q(\bar{\mathbf{x}}) \leftarrow \exists\bar{\mathbf{y}}.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{y}})$, and a vector $\bar{\mathbf{t}}$ of terms in universe(G), we say that $q(\bar{\mathbf{t}})$ is* entailed *by G, denoted $G \models q(\bar{\mathbf{t}})$, if and only if in any model $\mathcal{I}$ of G, there is a vector $\bar{\mathbf{t}}'$ of terms in universe(G) such that $\mathcal{I}$ is a model of the query instantiation $\varphi(\bar{\mathbf{t}}, \bar{\mathbf{t}}')$.*

Definition 3.17 (Query Answers)**.** *If $G \models q(\bar{\mathbf{t}})$ then $\bar{\mathbf{t}}$ is called an* answer *to q. The* answer set *of q w.r.t. G is defined as $ans(G, q) = \{\, \bar{\mathbf{t}} \mid G \models q(\bar{\mathbf{t}}) \,\}$.*

The notion of a solution for BGPs in SPARQL is the same as the notion of answers for conjunctive queries:

Proposition 3.1. *Given a graph G and a BGP P, the solutions of P are the same as the answers of the query $q(var(P)) \leftarrow P$, i.e. $ans(G, q) = [\![P]\!]_G$.*

SPARQL 1.1

A new version of SPARQL, called SPARQL 1.1 (Harris and Seaborne, 2012), is in the process of being proposed as a W3C recommendation. This new version is composed of several documents specifying the updated query language and introduces new features that were already used in practice by several SPARQL engines, such as: (i) aggregates; (ii) subqueries; (iii) negation; (iv) assignment; and (v) property paths. Other documents included in this new version, but not detailed in this section, specify an Update language (Gearon et al., 2012) and extensions for federated querying (Prud'hommeaux and Buil-Aranda, 2011).

Aggregates allow expressions to be applied over groups of solutions to obtain a single value, for example determining the minimum (`min`) value of the group. Other aggregator functions included in the standard are `count`, `sum`, `max`, `avg`, and `group_concat`. Although the use of aggregate functions was already available in several SPARQL engines, it will only be introduced into the official W3C specification with SPARQL 1.1.

[5] A predicate $p(\bar{\mathbf{x}}, y)$ is functional if for any $\bar{\mathbf{t}}$ there is a *unique* t' for which $p(\bar{\mathbf{t}}, t')$ is true.

In SPARQL 1.1, nested `select` queries are allowed to be used in graph patterns and the projected variables of the subquery are then joined with the results of the outer query. These nested `select` queries are however not allowed to specify a dataset and are restricted to be executed over the same dataset as the outer query. SPARQL follows a bottom-up query evaluation and thus the inner queries are evaluated first and its results made available to the outer query. A proposal for subqueries in SPARQL was previously presented by Angles and Gutiérrez (2010) and later the same authors compared different forms of subqueries to the W3C semantics (Angles and Gutiérrez, 2011).

Although negation was already permitted in SPARQL by using a combination of the `filter` and `bound` operators, this is made explicit in SPARQL 1.1 by allowing two forms of negation: the `exists` and `minus`. The `exists` (and `not exists`) `filter` expression allows to test if a graph pattern matches (or does not match) the dataset and consequently remove such solutions from the results. The other form of negation uses the `minus` operator that, when applied to two graph patterns, removes solutions from the left-hand side compatible with any solution from the right-hand side. Since the `minus` operator relies on the notion of compatible solutions, it will only remove solutions if there are shared variables between the solution sequences it is applied to. This causes different results between the two forms of negation when the provided graph patterns do not share variables:[6] since no two solutions are compatible, the `minus` operator does not remove any solutions from the resulting sequence. However, the `exists` operator will remove the respective solutions from the final sequence.

SPARQL 1.1 includes a basic query federation by means of the `SERVICE` keyword, which specifies that the following subquery will be executed in a remote SPARQL endpoint.

Other features include assignment of variables in the graph pattern (using the `bind` operator), in the `select` clause, and in the `group by` clause. All assignments are of the form "`(expression AS $var)`", where `expression` is the expression to be evaluated and `$var` is the variable name the result of the expression is assigned to. Another form of assignment is using the `bindings` clause,[7] which allows to specify a solution sequence that is to be joined with the results of the graph pattern. The values for variables in the provided solution sequence must be RDF terms, i.e. no variables can be specified. The `bindings` clause is envisioned to be used with the `service` keyword to specify values for federated querying.

Property paths are used to specify a connection between two RDF nodes. An extended graph pattern syntax is defined that allows for a concise pattern matching, for example specifying alternative routes for connecting the nodes, or to match paths of arbitrary or specific lengths.

3.4. Conclusion

This chapter introduced the SQL, XQuery, and SPARQL query languages that allow to access data in the formats presented in Chapter 2. Each query language focuses on a specific data model, namely SQL for relational data, XQuery for XML data, and SPARQL for RDF data. For XML, we briefly presented the XPath and XSLT languages, which are closely related to XQuery.

We briefly introduced the syntax and semantics of each language, with special focus on XQuery and SPARQL, which will be used in the next chapters to define the novel transformation language, called XSPARQL, and the extension of SPARQL towards querying meta-information. XSPARQL integrates the SQL, XQuery, and SPARQL query languages presented in this chapter, thus allowing to combine data from the different data models.

[6] A special case of graph patterns that do not share any variables is when the pattern to be removed contains no variables, i.e. is a fixed pattern.

[7] Note that the SPARQL 1.1 syntax is still under development and the `bindings` clause may be changed to `values`.

Part II.

Contributions

4. The XSPARQL Language

This chapter introduces a language that is capable of querying, transforming, and exposing data from heterogeneous sources, namely sources adhering to the data models presented in Chapter 2: the relational model, the tree-based XML and JSON formats, and the graph-based RDF model. This language, called XSPARQL, is based on the existing standard query languages described in Chapter 3: SQL, XQuery, and SPARQL, that are used to query the heterogeneous input sources. Since JSON does not specify a query language XSPARQL, automatically converts JSON into a predefined XML representation over which it is possible to use XQuery and XPath (this approach is detailed in Section 4.4). XSPARQL consists of an extension of the XQuery language with syntactical constructs from both SQL and SPARQL and as such XSPARQL is an XQuery-flavoured language, whose semantics is defined as an extension of the XQuery semantics. As a first example we can use this language to expose data in relational databases as RDF or XML data, in a similar approach to current proposals for translating relational data to RDF (RDB2RDF). But furthermore a common language including SQL, XQuery and SPARQL can support more involved transformations between different formats, for instance, enabling the integration of enterprise legacy data into LOD as described in Chapter 1. The importance of converting data between these data models has been acknowledged within the W3C in several standardisation efforts: Gleaning Resource Descriptions from Dialects of Languages (GRDDL) (Connolly, 2007), Semantic Annotations for Web Services Description Language (SAWSDL) (Farrell and Lausen, 2007), and more recently RDB2RDF (Arenas, Prud'hommeaux et al., 2012; Das, Sundara et al., 2012).

In data integration scenarios, such as the one described in Chapter 1, we often call the transformations from the different formats into RDF *lifting* and the transformations in the opposite direction *lowering*. The names derive from the fact that RDF is classified as having a higher abstraction level when compared to relational data or even semi-structured XML data.

Lifting: Transforming Heterogeneous Sources into RDF

Within the W3C, the GRDDL working group addressed the lifting task by allowing RDF data to be extracted from existing XML and (X)HTML Web pages. The XML or HTML document can link (by means of a specialised vocabulary) to XSLT transformations that, when applied to the original document, produce the RDF data. In the Web Services community, the Web Services Description Language (WSDL) (Chinnici et al., 2007) is an XML-based language for describing the messages that a web service accepts (sends and receives). The SAWSDL working group focused on defining mechanisms to add annotations to WSDL documents that allow the XML messages of a web service to be transformed into RDF (adhering to a specified schema) and, vice versa, enable the lowering of data stored in RDF and the creation of target XML messages. The ongoing RDB2RDF Working Group focuses on transforming data between the relational model and RDF, enabling the vast amounts of data contained in relational databases to be exposed as RDF, for example most enterprise data (as discussed in Chapter 1). The RDB2RDF Working Group has defined a mapping vocabulary that specifies how existing relational data can be converted into RDF. In Section 4.5 we will look at how the XSPARQL language implements this specification.

As described in Section 2.4.1, RDF/XML (Beckett, 2004) is the recommended syntax for RDF, using XML as the underlying representation model and, based on this format, it is conceivable to use XML-based

```
declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" ;
declare namespace foaf = "http://xmlns.com/foaf/0.1/" ;
declare namespace mo = "http://purl.org/ontology/mo/" ;
declare namespace dc = "http://purl.org/dc/elements/1.1/" ;

let $bandPos := distinct-values(//band/@name)
let $memberPos := distinct-values(//member)
let $albumPos := distinct-values(//album/@name)
let $songPos := distinct-values(//song)

return
 <rdf:RDF> {
  for $band in //band
  let $bandName := data($band/@name)
  let $bandID := fn:index-of($bandPos, $bandName)
  return
    <mo:MusicGroup rdf:nodeID="b{$bandID}" foaf:name="{$bandName}">{
      for $member in $band//member
        let $memberID := fn:index-of($memberPos, $member)
        return <foaf:member rdf:nodeID="m{$memberID}"/>
    }</mo:MusicGroup>,

  for $memberName at $memberID in $memberPos
  return <mo:MusicArtist rdf:nodeID="m{$memberID}">
           <foaf:name>{$memberName}</foaf:name>
         </mo:MusicArtist>,

  for $album in //album
  let $albumName := data($album//@name)
  let $albumID := fn:index-of($albumPos, $albumName)
  let $bandID := fn:index-of($bandPos, data($album/../../@name))
  return <mo:Record rdf:nodeID="a{$albumID}">
           <mo:title>{$albumName}</mo:title>
           <foaf:maker rdf:nodeID="b{$bandID}"/>{
             for $song in $album//song
             let $songID := fn:index-of($songPos, $song)
             return <mo:track rdf:nodeID="s{$songID}"/>
         }</mo:Record>,

  for $songName at $songID in $songPos
  return <mo:Track rdf:nodeID="s{$songID}">
           <dc:title>{$songName}</dc:title>
         </mo:Track>
 } </rdf:RDF>
```

Query 4.1: Lifting using XQuery

tools, such as XSLT or XQuery, to produce RDF data. Both the GRDDL and SAWSDL specifications use XSLT to perform lifting and lowering, however, as we will show, approaches that rely on RDF/XML for transformations between RDF and XML have several disadvantages. In the following examples we are using XQuery to perform the different transformations, similar transformations can also be achieved using XSLT but this does not invalidate any of the drawbacks we present.

Example 4.1 (Lifting in XQuery)**.** As an example of the lifting transformation, Query 4.1 presents the XQuery that converts the XML data from Data 2.1 into RDF. This query produces an RDF graph similar to the one presented in Data 2.4 with the exception that it uses blank nodes as idenfiers

for all entities, while the graph from Data 2.4 uses DBpedia URIs as identifiers. The blank node labels assigned to each entity are generated by using a prefix for each type of entity: **(b)**ands, **(m)**usic artists, **(a)**lbums, and **(s)**ongs, followed by a sequencial identifier (cf. `rdf:nodeID` in line 17). Having determined all the identifiers (in lines 6–9), the query produces the required RDF/XML structure: the triples refering to bands, their name and its members are generated in lines 13–21. A similar process is then repeated for artists (lines 23–26), albums (lines 28–38), and songs (lines 40–43).

While this example presents a valid solution for lifting, we can observe the following drawbacks:

- we have to build RDF/XML manually and cannot use the more readable and concise Turtle syntax; and
- the resulting RDF data is not guaranteed to be valid (according to Definition 2.2).

The task of lifting data from relational databases can be performed in a similar fashion by relying on SQL/XML (Eisenberg and Melton, 2001) however this would introduce an indirection step by first having to transform data into XML and then into RDF. Combining SQL, XQuery, and SPARQL in XSPARQL simplifies the lifting process, allowing to use SPARQL *ConstructClause*s to generate RDF in Turtle format (directly from relational data or XML) and performing automatic validation of the generated RDF.

Lowering: Transforming from RDF into the Legacy Formats

As we have seen the lifting task can be accomplished (with some drawbacks) by using XSLT or XQuery. On the other hand, converting from RDF data back into the legacy data models using XML tools poses obstacles that are even harder to overcome, namely:

- the flexibility of the RDF/XML format (and the lack of a canonical format) makes writing transformations difficult;
- merging different RDF graphs may involve complex processing (e.g. renaming of blank nodes); and
- possibly handling the interplay with inference mechanisms e.g. RDFS would require custom-built code.

As we have presented in Section 2.4.1, the RDF/XML serialisation format is very flexible, as it includes several shortcuts and allows for different representations of the same RDF graph. For example, we have shown two equivalent serialisations for the same RDF graph in Data 2.3 and 2.4, however both serialisations are very different when we focus on their XML structure. Since using XML tools requires handling RDF/XML as XML data, all the possible different serialisations for an RDF graph would need to be taken into account.

Example 4.2 (Lowering in XQuery)**.** Query 4.2 performs the lowering task directly from RDF/XML using XQuery. This query first retrieves all the `mo:Band` XML elements (lines 8–23) and, for each band, retrives the names of the artists (lines 11–14) and albums (lines 17–21) of the band. Furthermore, all the song names of each album are collected in lines 19–20. This process creates the desired nested structure of the XML file presented in Data 2.1.

One issue is that Query 4.2 is tailiored specifically to the RDF/XML serialisation from Data 2.4 and will not produce the desired results if the serialisation changes. Although creating a query capable of handling any RDF/XML serialisation would be possible, this would be a cumbersome and error-prone task. Futhermore, if the RDF data is stored in the Turtle serialisation it is not possible to use XML tools.

On the other hand, SPARQL is agnostic to the actual XML representation of the underlying source graphs, which alleviates the pain of having to deal with different RDF/XML representations of the graphs. Also merging several RDF source graphs specified in consecutive `from` clauses (as described in Section 3.3),

```
declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" ;
declare namespace foaf = "http://xmlns.com/foaf/0.1/" ;
declare namespace mo = "http://purl.org/ontology/mo/" ;
declare namespace dc = "http://purl.org/dc/elements/1.1/" ;

<user>
  <bands>{
  for $band in //mo:MusicGroup
  return <band name="{$band/@foaf:name}">
          <members>{
            for $member in $band/foaf:member
            return <member>{
              //mo:MusicArtist[@rdf:about = $member/@rdf:resource]/foaf:name/text()
            }</member>
          }</members>
          <albums>{
            for $album in //mo:Record[./foaf:maker/@rdf:resource = $band/@rdf:about]
            return <album name="{$album//mo:title}">{
                for $song in $album/mo:track/@rdf:nodeID
                return <song>{//mo:Track[@rdf:nodeID = $song]/dc:title/text()}</song>
              }</album>
          }</albums>
        </band>
}</bands></user>
```

Query 4.2: Lowering using XQuery

which could involve renaming of blank nodes at the pure XML level, comes for free in SPARQL. However, we cannot use SPARQL alone for the lowering transformations since SPARQL does not provide the possibility of handling XML data.

Apart from its syntactic ambiguities, processing RDF/XML via XQuery also loses another feature of RDF, namely its interplay with ontological information, e.g. RDFS. Since XML tools do not support ontological inference, we would need to implement an RDFS inference engine within XSLT or XQuery, to cater for a lowering mechanism that also works for this kind of RDF data. Given the availability of RDF tools and engines that readily offer RDFS support via materialising inferences, this is a dispensable exercise. Furthermore, in Chapter 6 we present an extension of the RDFS inference rules (called Annotated RDFS) and of the SPARQL query language towards meta-information and in Chapter 7 we present the combination of XSPARQL and Annotated RDFS, which also introduces inferencing capabilities in XSPARQL.

Benefits of an Integrated Language

In recognition of the above problems the SAWSDL specification contains a non-normative example, which performs a lowering transformation by applying an XSLT transformation to the XML representation of the results of a SPARQL query (Clark et al., 2008). Such a two-step approach alleviates the issues described: first, since SPARQL works on the RDF data model the different RDF/XML serialisations are considered to be equivalent, and second, RDFS inferences can also be catered for in the SPARQL engine. Although the approach proposed by the SAWSDL Working Group provides a good starting point, it can still be improved on several points: firstly, the detour through SPARQL's XML query results format is an unnecessary burden. Secondly, a more tightly-coupled integration of the different query languages can provide a more expressive language, beyond the capabilities of using different languages sequentially, and directly amenable to query optimisations. The proposed language, XSPARQL, aims to provide exactly this: use cases that otherwise would require interleaved calls to SPARQL (typically

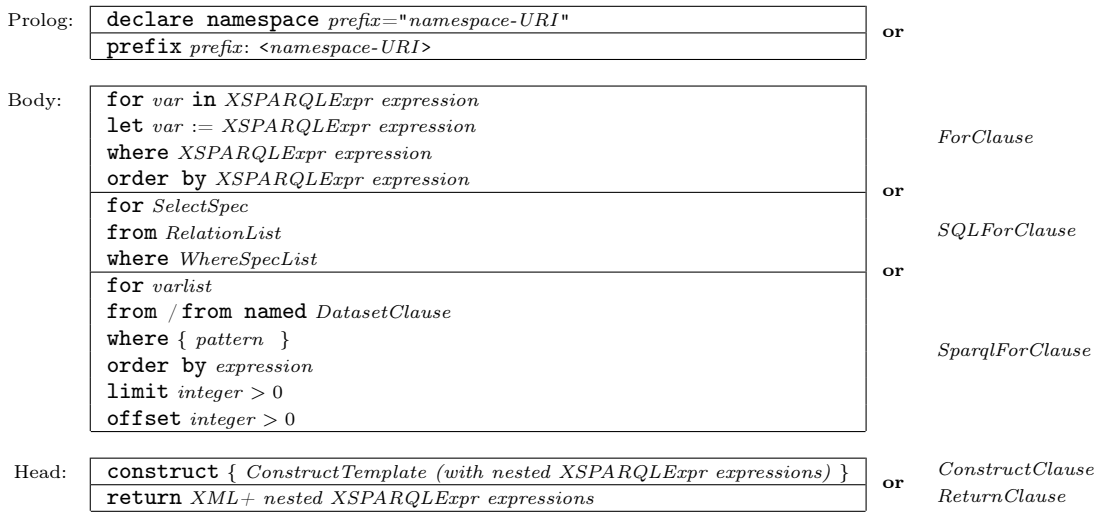


Figure 4.1.: Schematic view of XSPARQL

requiring an implementation using an external programming framework) can be solved in XSPARQL directly, cf. the lowering example in Query 4.3.

In fact, the current version of SPARQL (Prud'hommeaux and Seaborne, 2008) is still preliminary in terms of expressivity when compared to SQL or XQuery. As a side effect of this integration, XSPARQL also extends SPARQL's expressiveness for pure RDF transformations by allowing, for instance, nested XSPARQL queries in the graph construction step. The SPARQL 1.1 query language (cf. Section 3.3), currently under development by the W3C SPARQL Working Group, gives a leap forward in terms of expressivity, however, providing mechanisms to convert data back into the native legacy data models of XML or SQL databases is beyond the scope of the Working Group.

We next present the syntax and semantics of the XSPARQL language in Sections 4.1 and 4.2, respectively. We continue by presenting properties relating the novel language with its constituent languages (Section 4.3), introduce our handling of JSON data in Section 4.4, and presenting the processing of RDB2RDF mappings within XSPARQL (Section 4.5). Finally, we introduce a discussion on related works in Section 4.6.

4.1. Syntax

Conceptually, XSPARQL is a merge of XQuery, SPARQL `construct` and `select` queries, and SQL `select` queries, as presented schematically in Figure 4.1. This re-use of different query languages allows us to benefit from their facilities for retrieving data in the different models, while also allowing us to use Turtle-like syntax for constructing RDF graphs (inherited from the SPARQL language). Since XSPARQL is based on XQuery, we allow any native XQuery expression and further extend XQuery's syntax with the following expressions:

(i) XQuery and SPARQL namespace declarations in the Prolog may be used interchangeably;
(ii) in the Body, we allow the existing XQuery *ForClause*s and also SPARQL `select` queries (*SparqlForClause*) and SQL `select` queries (*SQLForClause*); and
(iii) in addition to XQuery's native *ReturnClause*, in the head we allow RDF graphs to be created directly using `construct` templates (*ConstructClause*).

In XSPARQL we also allow different forms of nesting: (i) `let` assignments can contain the result of subqueries that construct RDF graphs, and the assigned variable can later be used in SPARQL-style `from`

XSPARQLExpr	::=	(*FLWOExpr* | *SQLForClause* | *SparqlForClause*) (*ReturnClause* | *ConstructClause*)
FLWOExpr	::=	(*ForClause* | *LetClause*)+ *XQWhereClause*? *OrderByClause*?
ReturnClause	::=	'`return`' *ExprSingle*
SQLForClause	::=	'`for`' *SelectSpec RelationList SQLWhereClause*?
SparqlForClause	::=	'`for`' (*VarName*+ | '`*`') *DatasetClause*? *WhereClause*? *SolutionModifier*
ConstructClause	::=	'`construct`' *ConstructTemplate'*

Figure 4.2.: *XSPARQLExpr* syntax overview

clauses, or (ii) nesting can also be used for value construction within SPARQL-style `construct` templates. Since the new *SQLForClause* and *SparqlForClause* expressions stand at the same level as XQuery's `for` and `let` expressions, such clauses are allowed to start new *XSPARQLExpr* expressions and may also occur inside deeply nested XSPARQL queries. The main difference between these new expressions and SQL and SPARQL `select` expressions is that while the latter expressions return bindings for variables (as described in Sections 3.1.2 and 3.3), the new expressions follow an approach similar to XQuery's *ForClause* by adding new variables to the scope of query and as such we choose a syntax also inspired by the XQuery *ForClause*.

An overview of the grammar productions for these newly introduced expressions (*SQLForClause*, *SparqlForClause*, and *ConstructClause*) is presented in Figure 4.2. Notably, when compared to the XQuery grammar, we introduced a new production (*XSPARQLExpr*) that changes the XQuery *FLWORExpr* to include the new expressions.

We next look at the syntax of each newly introduced expression in more detail while presenting some XSPARQL query examples that allow us to perform the lifting and lowering tasks in a straightforward fashion.

4.1.1. *SparqlForClause*

SparqlForClause ::= '`for`' (*VarRef*+ | '`*`') *DatasetClause*? *WhereClause*? *SolutionModifier*

The newly introduced *SparqlForClause* is similar to an XQuery `for` expression that returns a sequence of SPARQL results. In this grammar production, the *WhereClause* and *SolutionModifier* correspond to rules [13] and [14] from the SPARQL grammar, respectively cf. Prud'hommeaux and Seaborne (2008, Appendix A.8). Similar to SPARQL's and SQL's '`select *`' shortcut, we allow to write '`for *`' in place of '`for [`*list of all unbound variables appearing in the WhereClause*`]`' in *SparqlForClause*s, which effectively avoids listing the distinguished variables of the query.

We also extended the rules for the SPARQL `SourceSelector` grammar expression (rule [12] of the SPARQL grammar) in order to allow graphs in a dataset to be specified by a variable:

SourceSelector ::= *IRIref* | *VarRef*

The variables used here must contain an RDF graph, resulting from a *ConstructClause* (as described in the next section and further detailed in Section 4.2).

Regarding the syntax for variables in XQuery and SPARQL, we restrict the use of SPARQL '`?`'-prefixed variables and allow only '$'-prefixed variables that are compatible with XQuery's variable specifications. On the other hand, as mentioned in Section 3.2.3, XQuery also allows to specify variables as QNames, allowing the disambiguation of variables based on their namespace. However, since such variable names are not allowed in SPARQL we further assume that only unprefixed variables are shared between the XQuery and SPARQL expressions of XSPARQL.

```
1  declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" ;
2  declare namespace foaf = "http://xmlns.com/foaf/0.1/" ;
3  declare namespace mo = "http://purl.org/ontology/mo/" ;
4  declare namespace dc = "http://purl.org/dc/elements/1.1/" ;
5
6  <user><bands>{
7    for * from <bands.ttl>
8    where { $band a mo:MusicGroup ; foaf:name $bandName . }
9    return <band name="{$bandName}">
10           <members>{
11             for $memberName from <bands.ttl>
12             where { $band foaf:member $bandMember .
13                     $bandMember foaf:name $memberName . }
14             return <member>{$memberName}</member>
15           }</members>
16           <albums>{
17             for * from <bands.ttl>
18             where { $album foaf:maker $band .
19                     $album mo:title $albumName . }
20             return <album name="{$albumName}">{
21               for * from <bands.ttl>
22               where { $album mo:track $song .
23                       $song dc:title $songName . }
24               return <song>{$songName}</song>
25             }</album>
26           }</albums>
27         </band>
28 }</bands></user>
```

Query 4.3: Lowering using XSPARQL

The lowering transformation can also be rewritten using XSPARQL. These are the kind of transformations that present extra problems for the XSLT and XQuery languages and where we can see the advantages of using XSPARQL. By using the introduced *SparqlForClause*s for accessing the RDF graph, XSPARQL avoids handling RDF as XML data, along with all the encapsulated issues.

Example 4.3 (Lowering RDF data with XSPARQL)**.** The lowering XSPARQL query for our running example is shown in Query 4.3. Here we can note the inclusion of *SparqlForClause*s, for instance in line 7, to retrieve all the bands (`mo:Band`) contained in the RDF data. Furthermore, nested *SparqlForClause*s can be used for further processing of the input data: the *SparqlForClause* in lines 11–13 is responsible for retrieving all the members of the respective band, where the '`$band`' variable is instantiated with the band identifier currently being processed. A similar structure is repeated for converting the corresponding albums of the band (lines 17–26) and songs of each album (lines 21–24).

4.1.2. *ConstructClause*

ConstructClause ::= '`construct`' *ConstructTemplate*

The *ConstructClause* allows XSPARQL to produce RDF graphs and, by following SPARQL's restrictions on the generated RDF triples (cf. Section 3.3), we also ensure that the resulting graph is valid RDF. The XSPARQL *ConstructTemplate* expression is defined in the same way as the production *ConstructTemplate* in SPARQL (Prud'hommeaux and Seaborne, 2008), but we additionally allow nested *XSPARQLExpr* expressions in subject, predicate, and object positions. We allow three types of nested expressions, identified by the shortcuts '`{`' *XSPARQLExpr* '`}`', '`<{`' *XSPARQLExpr* '`}>`', and '`_:{`' *XSPARQLExpr* '`}`'

```
1  declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" ;
2  declare namespace foaf = "http://xmlns.com/foaf/0.1/" ;
3  declare namespace mo = "http://purl.org/ontology/mo/" ;
4  declare namespace dc = "http://purl.org/dc/elements/1.1/" ;
5
6  let $bandPos := distinct-values(//band/@name)
7  let $memberPos := distinct-values(//member)
8  let $albumPos := distinct-values(//album/@name)
9  let $songPos := distinct-values(//song)
10
11 return (
12   for $band in //band
13   let $bandName := data($band/@name)
14   let $bandID := fn:index-of($bandPos, $bandName)
15   construct { _:b{$bandID} a mo:MusicGroup ; foaf:name {$bandName} .
16               { for $member in $band//member
17                 let $memberID := fn:index-of($memberPos, $member)
18                 construct { _:b{$bandID} foaf:member _:m{$memberID} }
19               } },
20
21   for $memberName at $memberID in $memberPos
22   construct { _:m{$memberID} a mo:MusicArtist; foaf:name {$memberName} },
23
24   for $album in //album
25   let $albumName := data($album//@name)
26   let $albumID := fn:index-of($albumPos, $albumName)
27   let $bandID := fn:index-of($bandPos, data($album/../../@name))
28   construct { _:a{$albumID} a mo:Record; mo:title {$albumName};
29                             foaf:maker _:b{$bandID} .
30                             { for $song in $album//song
31                               let $songID := fn:index-of($songPos, $song)
32                                 construct { _:a{$albumID} mo:track _:s{$songID} } }
33             },
34
35   for $songName at $songID in $songPos
36   construct { _:s{$songID} a mo:Track; dc:title {$songName} }
37 )
```

Query 4.4: Lifting in XSPARQL

that construct *literals*, *URIs*, and *blank nodes*, respectively. This syntax is used during static analysis to correctly determine the type of each element: `literal`, `uri`, and `bnode` (cf. Section 4.2.1 below).

Additionally, we also allow SPARQL-style *ConstructClause*s to appear before the body part of queries, and as such XSPARQL becomes a syntactic superset of native SPARQL `construct` queries (with the minor exception being the restriction on '`?`'-prefixed variables).

The following lifting query shows the use of the *ConstructClause* expression.

Example 4.4 (Lifting XML data with XSPARQL)**.** Query 4.1 can be reformulated into its slightly more concise XSPARQL version in Query 4.4. This query behaves in a similar way to Query 4.1, creating the RDF triples for each entity in the input XML data. The difference is that we are using nested SPARQL-like `construct` clauses for creating the RDF triples (cf. lines 15–19). In line 36 we use the different XSPARQL shortcuts, in this case to create URIs and literals. The result of this query is also guaranteed to be valid RDF as explained in Section 4.2.4.

4.1.3. *SQLForClause*

SQLForClause ::= '`for`' *SelectSpec RelationList SQLWhereClause?*

The *SQLForClause* element represents an SQL `select` query that can be evaluated against the underlying database. Similar to XQuery's `for` clause, the *SQLForClause* expression represents the results of the execution of a SQL query and exposes the result values to other subsequent expressions in the query. The additional *SQLForClause* syntax rule are presented next, where *VarRef* corresponds to an XSPARQL variable ('$'-prefixed), *TableAlias* represents a string used as an alternative name for the relation, and *Constant* represents an integer or string:

SelectSpec ::= *AttrSpecList* | '`*`' | '`row`' *VarRef*

AttrSpecList ::= *AttrSpec AttrNameSpec?* ('`,`' *AttrSpec AttrNameSpec?*)*

AttrSpec ::= *attrName*
| *relationName* '`.`' *attrName*
| *VarRef*

AttrNameSpec ::= '`as`' *VarRef*

RelationList ::= '`from`' *RelationSelector* ('`,`' *RelationSelector*)*

RelationSelector ::= *RelationName* ('`as`' *RelationAlias*)?
| *VarRef* ('`as`' *RelationAlias*)?

SQLWhereClause ::= '`where`' *WhereSpecList*

WhereSpecList ::= '`(`' *WhereSpecList BooleanOp WhereSpecList* '`)`'
| *WhereSpec*

WhereSpec ::= *WhereAttrSpec ComparisonOp WhereAttrSpec*
| *WhereAttrSpec ComparisonOp Constant*
| *Constant ComparisonOp WhereAttrSpec*

WhereAttrSpec ::= *AttrSpec*
| '`{`' *VarRef* '`}`'

BooleanOp ::= '`and`' | '`or`'

ComparisonOp ::= '`=`' | '`!=`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`=>`'

When comparing the XQuery and SQL languages we find a syntactical mismatch between the representation of variables: while SQL considers the relation names specified in *RelationSelector* as variables (as described in Section 3.1.2), XQuery assumes '`$`'-prefixed variable names. XSPARQL provides ways of overcoming this mismatch, allowing to specify variable names for the results of an *SQLForClause*, by:

(i) explicitly specifying a variable name for each attribute – represented by the syntax rule *AttrNameSpec*, where *VarRef* is the variable name to which the attribute value is assigned: e.g. '`for bands.bandId as $bandId`';
(ii) implicitly by omitting the variable name or using '`for *`'; and
(iii) using the '`row`' keyword instantiates the specified variable with each *result row* the query produces.

For (ii), each attribute in the result set is assigned a variable name automatically with the same name as the attribute name, of the format: '`$relationName.attributeName`'.

Example 4.5 (Variable Name Generation)**.** Consider the relational schema presented in Example 2.2. If we specify a *SQLForClause* in the form of '`for * from person`', the variable names that will be available for the query will be '`$person.personId`', '`$person.personName`', and '`$person.bandId`'.

```
declare namespace rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#" ;
declare namespace foaf = "http://xmlns.com/foaf/0.1/" ;
declare namespace mo = "http://purl.org/ontology/mo/" ;
declare namespace dc = "http://purl.org/dc/elements/1.1/" ;

(
  for band.bandId as $bandID, band.bandName as $bandName
  from band
  construct { _:b{$bandID} a mo:MusicGroup ; foaf:name {$bandName} },

  for person.personId as $memberID, person.personName as $memberName, person.bandId as
       $bandID
  from person
  construct { _:m{$memberID} a mo:MusicArtist; foaf:name {$memberName} .
              _:b{$bandID} foaf:member _:m{$memberID}
            },

  for album.albumId as $albumID, album.albumName as $albumName, album.bandId as
      $bandID
  from album
  construct { _:a{$albumID} a mo:Record; mo:title {$albumName};
                           foaf:maker _:b{$bandID} .
            },

  for song.songId as $songID, song.songName as $songName, song.albumId as $albumID
  from song
  construct { _:s{$songID} a mo:Track; dc:title {$songName} .
              _:a{$albumID} mo:track _:s{$songID}
             }
)
```

Query 4.5: Lifting from relational database

If the relation attributes are not known beforehand, e.g. if the relation is specified as a variable, it is not possible to generate the variable names as described in (ii). In this case, we can use '`row $r`' in place of the variable names specification, and at execution time, '`$r`' will be instantiated with an XML representation containing all the attributes in the queried relations. It is then possible to access all the attributes or to retrieve (if known) a specific attribute. This form of selecting attributes is necessary for processing RDB2RDF mappings (presented in Section 4.5.3) since the queried relations and attributes are read from a user-specified RDF graph and thus the attributes of the relation cannot be determined during syntactical analysis of the query.

In SQL, `where` clauses indicate specific values of an attribute to be matched or that the value of two attributes must be the same. When we introduce the extended XSPARQL syntax (which allows to use $-prefixed variables) we need a way to specify if the variable represents an attribute name or an attribute value. We make this distinction in the syntax of XSPARQL: a $-prefixed variable represents an attribute value, in case we want a variable to represent an attribute name of a relation we use the '`{`' *VarRef* '`}`' syntax. Further details on how XSPARQL handles this distinction are presented in Section 4.2.

In a similar fashion to the lifting query from XML data (Query 4.4), we can use *SQLForClause*s to access relational data and convert it to RDF, as presented in the following example.

Example 4.6 (Lifting Relational data with XSPARQL)**.** Query 4.5 shows an XSPARQL query that performs the lifting task over the relational schema described in Example 2.2. In this query we are using the primary key (generated identifier) of each relation for generating the blank node label of

each entity (cf. line 9). The rest of the query consists of creating the respective RDF triples for the other relations: person (lines 11–15), album (lines 17–21), and song (lines 23–27).

4.2. Semantics

Next we define the semantics of XSPARQL by reusing the semantics of SQL and SPARQL. We start by defining how we ensure that the semantics of SQL and SPARQL queries respects any existing XSPARQL variable bindings. In Section 4.2.3, we present the extensions to the W3C XQuery's semantics (Draper, Fankhauser et al., 2010), namely the new types we use, an extension to the normalisation rules of XQuery *ForClause*s, and necessary additional environment components. Section 4.2.4 presents the semantics of the newly introduced expressions: *SparqlForClause*, *ConstructClause*, and *SQLForClause*, based on XQuery's formal semantics (Draper, Fankhauser et al., 2010), by defining normalisation, static type and dynamic evaluation rules for each of the new expressions.

4.2.1. XSPARQL Types

We extend the XQuery 1.0 and XPath 2.0 Data Model (described in Section 2.2.3) with the following new types to accommodate for SQL and SPARQL specific parts of XSPARQL:

(1) the `SQLTerm` is an extension of `xs:anyAtomicType` (as presented in Section 3.1.2);
(2) the `RDFTerm` type further consists of the subtypes `uri`, `bnode` and `literal` and is used as the type of SPARQL variables;
(3) the `PatternSolution` type consists of a sequence of pairs (*variableName*, `RDFTerm`), representing SQL or SPARQL variable bindings;
(4) the `RDFGraph` is the type resulting from the evaluation of `construct` expressions; and
(5) the `RDFDataset` is the type used for representing RDF datasets, which is further constituted by one default `RDFGraph` and a sequence of named graphs (`RDFNamedGraph`).

Figure 4.3 presents the formal definition of (1)–(5) following the notation for XML Schema datatypes (presented in Section 2.2.2). The `RDFTerm` type is used to represent RDF terms (composed of URIs, blank nodes or literals). The type of SPARQL variables is represented by the `Binding` type, that consists of the variable name and the RDF term that is assigned to it. Finally, sequences of SPARQL variable bindings are represented by the type `PatternSolution`.

Similarly for SQL results, sequences of SQL variable bindings are also represented by the type `PatternSolution`. Analogously, we define the types `SQLResult` and `SQLBinding` for representing SQL results. The `SQLBinding` type is defined as an extension of `xs:anyAtomicType`, and we follow the mapping from SQL types into XML types presented in Table 3.1.

The `RDFGraph` type corresponds to a sequence of `RDFTriples`, which are in turn a complex type composed of `subject`, `predicate` and `object`. The `RDFDataset` type is defined as an `RDFGraph` that is considered the default graph and a sequence of `RDFNamedGraphs` represented by the `name` of the graph and the corresponding `RDFGraph`.

Translating SQL and SPARQL Solutions into the `PatternSolution` Type

The next definition presents the translation between a SPARQL solution sequence and a sequence of `SPARQLResult` type elements that we implement in XSPARQL. This serialisation of SPARQL results mimics the SPARQL Query Results XML Format (Beckett and Broekstra, 2008), defined by the XML Schema available at `http://www.w3.org/2007/SPARQL/result.xsd`.

```
define type URI-reference restricts xs:anyURI;

define type Literal extends xs:string {
      attribute datatype of type URI-reference?,
      attribute lang of type xml:lang? };

define type RDFTerm {
      element uri of type URI-reference |
      element bnode of type xs:string |
      element literal of type Literal };

define type SPARQLBinding extends RDFTerm {
      attribute name of type xs:string };

define type SPARQLResult {
      element binding of type SPARQLBinding* };

define type SQLTerm extends xs:anyAtomicType ;

define type SQLBinding extends SQLTerm {
      attribute name of type xs:string };

define type SQLResult {
      element binding of type SQLBinding* };

define type SQLAttribute extends xs:string ;

define type PatternSolution {
      element result of type SPARQLResult |
      element result of type SQLResult };

define type RDFGraph {
      element triple of type RDFTriple* };

define type RDFTriple {
      element subject of type RDFTerm,
      element predicate of type RDFTerm,
      element object of type RDFTerm };

define type RDFDataset {
      element defaultGraph of type RDFGraph,
      element namedGraphs of type RDFNamedGraphs };

define type RDFNamedGraphs {
      element namedGraph of type RDFNamedGraph* };

define type RDFNamedGraph {
      attribute name of type xs:string,
      element graph of type RDFGraph };
```

Figure 4.3.: XSPARQL Type Definitions

Definition 4.1 (Serialisation of Solution Sequences)**.** *Given a SPARQL solution sequence* $\Omega = (\mu_1, \ldots, \mu_n)$ *a serialisation of* Ω *into a sequence of* `PatternSolution` *is defined as follows:*

- $serialise(\Omega) \Rightarrow serialise(\mu_1), \ldots, serialise(\mu_n)$
- $serialise(\mu) \Rightarrow$ `<result>`$\{\forall x \in dom(\mu), serialise(\mu, x)\}$`</result>`
- $serialise(\mu, x) \Rightarrow$ `<binding name="`x`">`$\{term(\mu(x))\}$`</binding>`, *where* $term(\mu(x))$ *is*
  - `<uri>`$\{\mu(x)\}$`</uri>` *if* $\mu(x) \in \mathbf{U}$
  - `<bnode>`$\{\mu(x)\}$`</bnode>` *if* $\mu(x) \in \mathbf{B}$
  - `<literal>`$\{\mu(x)\}$`</literal>` *if* $\mu(x) \in \mathbf{L}$

Following the definition of the *serialise* function, in evaluation rules, we will refer to sequences of elements of type `PatternSolution` as Ω and to elements of type `SPARQLResult` as μ.

For the representation of SQL results we follow a similar approach:

Definition 4.2 (Serialisation of SQL Relation Instances)**.** *The serialisation of a relation instance* $I = (I_1, \ldots, I_n)$ *of relation* R *with* $sort(R) = U$*, into* `PatternSolution` *is:*

- $serialise(I) \Rightarrow serialise(I_1), \ldots, serialise(I_n)$
- $serialise(I_i) \Rightarrow$ `<result>`$\{\forall x \in U, serialise(I_i, x)\}$`</result>`
- $serialise(I_i, x) \Rightarrow$ `<binding name="`x`">`$\{sql2xml(I_i(x))\}$`</binding>`.

Serialisation into SQL and SPARQL Representations

The following definitions present the *SQLTerm* and *RDFTerm* functions that, when applied to an XSD datatype, return their representation in SQL or SPARQL syntax, respectively. We first present the serialisation into SQL:

Definition 4.3 (SQL representation)**.** *Let* C *be an expression context with static environment* $T_C = \text{statEnv}(C)$ *and dynamic environment* $D_C = dynEnv(C)$*, and* $x \in dom(T_C.varType)$ *an XSPARQL variable name. The SQL representation of* x *according to* C*, denoted* $SQLTerm_C(x)$ *is:*

- $data(D_C.varValue(x))$ *if* $T_C.varType(x) =$ (`SQLTerm` *or* `SQLAttribute` *or* `RDFTerm` *or* `node()`)*; and*

- *xml2sql*(D_C.*varValue*(x)) *otherwise,*

where xml2sql is the value conversion function presented in Section 3.1.2.

Similarly, we next present the serialisation of SPARQL terms:

Definition 4.4 (*RDFTerm*)**.** *Let C be an expression context with static environment $T_C = \text{statEnv}(C)$ and dynamic environment $D_C = dynEnv(C)$, and $x \in dom(T_C.varType)$ an XSPARQL variable name. The RDF representation of x according to C, denoted $RDFTerm_C(x)$ is:*

- D_C.*varValue*(x) *if* T_C.*varType*(x) = `RDFTerm`,
- *"*D_C.*varValue*(x) *"* *if* T_C.*varType*(x) = `xsd:string`,
- *"*D_C.*varValue*(`x`) `"^^rdf:XMLLiteral` *if* T_C.*varType*(x) = `element()`,
- *"data*(D_C.*varValue*(x))*"* *if* T_C.*varType*(x) = (`attribute()` *or* `SQLTerm` *or* `SQLAttribute`), *and*
- *"*D_C.*varValue*(x) `"^^`T_C.*varType*(x) *otherwise.*

4.2.2. XSPARQL Semantics for Querying Relational and RDF data

We now define the semantics of *SQLForClause*s and *SparqlForClause*s by relying on the evaluation semantics of their original query languages, namely SQL (presented in Section 3.1.2) and SPARQL (presented in Section 3.3). The approach we take is to rely on the translation of each language into their respective algebra expressions and further combine these algebra expressions with any existing XSPARQL variable bindings. Since XSPARQL is based on the semantics of XQuery, variable bindings are stored in the varValue environment component of the dynamic environment (cf. Section 3.2.3), that maps variable names to their value. Next we present how we interpret these variable mappings as a relation and as a solution sequence, thus allowing to combine the results of SQL and SPARQL queries with the existing variable bindings.

Querying Relational Data

In order to reuse the semantics of SQL for defining the semantics of XSPARQL *SQLForClause*s we transform the varValue component of the dynamic environment in which the *SQLForClause* is executed into a relation (which we call the *XSPARQL instance relation*). The following definition presents this translation:

Definition 4.5 (XSPARQL instance relation)**.** *Let the set of relation names (*$\mathbf{R}$*) be defined as in Section 2.1, and let C be an expression context. The* XSPARQL instance relation *of C is a relation instance named 'xir_C', where xir_C is a reserved relation name, i.e. $xir_C \notin \mathbf{R}$, and $sort(xir_C) = dom(dynEnv(C).varValue)$. For each mapping $v_i \to x_i \in dynEnv(C).varValue$, the value of xir_C for attribute v_i, denoted $xir_C(v_i)$, is defined as:*

- *if $x_i = ()$ is an empty sequence then $xir_C(x_i) =$* `null`*;*
- *if $x_i = (e_1, \ldots, e_n)$ is a sequence, then $xir_C(x_i) = fn{:}concat(SQLTerm_C(e_1), \cdots, SQLTerm_C(e_n))$.*[1]

For a SQLWhereClause S, we call the XSPARQL instance relation of the expression context in which S is executed the XSPARQL instance relation *of S.*

Another necessary step to enable the reuse of SQL evaluation semantics is to convert our extended syntax (that allows for $-prefixed variable names) into valid SQL syntax: each *WhereSpec* in a *SQLForClause* that contains an XSPARQL variable is removed from the normalised SQL query (by replacing it with '`true`') and is stored for a later evaluation by the XSPARQL semantics. For this we rely on the following normalisation function:

[1] Since the values of any relation attribute must be atomic, in the case of a variable being assigned to an XQuery sequence we assume the concatenation of each element of the sequence.

Definition 4.6 (SQL Representation of *SQLWhereClauses*)**.** *Let* S = '`where`' *WhereSpecList be a SQLWhereClause. The normalisation of* S*,* $normaliseSQL(S)$ = '`where`' $normaliseSQL($*WhereSpecList*$)$*, where* $normaliseSQL($*WhereSpecList*$)$ *is defined as:*

- *if WhereSpecList is of form* '`(`' $WhereSpecList_1$ *Op* $WhereSpecList_2$ '`)`' *then*

$$\text{'(' } normaliseSQL(WhereSpecList_1) \text{ } Op \text{ } normaliseSQL(WhereSpecList_2) \text{ ')'}$$

- *if WhereSpecList is of form* $Attr_1$ *Op* $Attr_2$ *then* $normaliseSQL(Attr_1 \text{ } Op \text{ } Attr_2)$ *is:*

$$\begin{cases} \text{'true'} & \text{if } Attr_1 \text{ or } Attr_2 \text{ is an XSPARQL variable} \\ Attr_1 \text{ } Op \text{ } Attr_2 & \text{otherwise.} \end{cases}$$

Furthermore we denote the set of WhereSpec of S *in which an attribute is an XSPARQL variable as* $whereSpecVars(S)$*.*

The normalisation of complete *SQLForClause*s consists also of the normalisation of the syntactical elements *AttrSpecList* and *TableSelector* presented in Section 4.1.3. In the normalisation of *AttrSpecList* we remove any existing *AttrNameSpec* component, since they reflect only the name of the corresponding XSPARQL variable. However, the normalisation of the *TableSelector* can only be performed during the dynamic evaluation of the XSPARQL query since any variables present in the *TableSelector* must be evaluated to determine the corresponding relation name. With the restriction of performing the substitution at evaluation time, we can reuse the standard translation of a SQL query into relational algebra as presented in Section 3.1.2.

Next we present how XSPARQL combines the results of a SQL query with an XSPARQL instance mapping. For this we rely on the standard relational selection (σ) and cross-product ($\times$) algebra operators presented in Section 3.1 and on the xir_C relation instance from Definition 4.5. Firstly, we present the construction of the relational algebra select expression that, based on the provided *SQLForClause* S and the XSPARQL instance mapping of S, makes the connection between the results of the SQL query and the existing XSPARQL variable bindings:

Definition 4.7 (XSPARQL σ expression)**.** *Let* S *be a SQLForClause with expression context* C *and* $V = whereSpecVars(S)$ *the attribute specifications that contain XSPARQL variables in* S*. The* XSPARQL σ expression *of* S*, denoted* $\sigma_{xs}(S)$*, is a relational algebra* σ *expression that, for each* Att_1 *Op* $Att_2 \in V$ *is* $trans(Att_1)$ *Op* $trans(Att_2)$*, where* $trans(Att)$ *is defined as:*

- *Attr* *if Attr is not an XSPARQL variable;*
- *if* $Attr$ = '`$`'$AttrName$ *is an XSPARQL variable then*

$$trans(Attr) = \begin{cases} dynEnv.varValue(AttrName) & \text{if statEnv}.varType(Attr) = \texttt{SQLAttribute} \\ \text{'}xir_C.AttrName\text{'} & \text{otherwise.} \end{cases}$$

This definition creates a relational algebra expression from the extended XSPARQL *SQLForClause* syntax, which can then be used to further restrict the results of the normalised SQL expression. Based on these definitions we can introduce the translation of *SQLForClause*s into relational algebra.

Definition 4.8 (XSPARQL relational algebra expression)**.** *Let* Q *be a SQLForClause,* $Q' = normaliseSQL(Q)$ *the SQL rewriting of* Q*,* $E = \sigma_{xs}(S)$ *the XSPARQL* σ *expression of* S*, and* $RA_{sql}(Q')$ *the relational algebra expression obtained from the standard SQL translation into relational algebra. The* XSPARQL relational algebra expression *of* Q*, denoted* $RA_{xsp}(Q)$*, combines the relational algebra expression of the SQL query and restricts its results to the existing bindings for XSPARQL variables as follows:*

$$\sigma_E(RA_{sql}(Q') \times xir_C) \quad .$$

```
1 let $x := 1
2 for bandName from band
3 where bandId = $x
4       and bandName = 'Nightwish'
5 return $bandName
```

(a) Value Matching

```
1 let $x := "bandName"
2 for bandName from band
3 where bandId = 1
4       and {$x} = 'Nightwish'
5 return $bandName
```

(b) Attribute Matching

Figure 4.4.: XSPARQL *SQLForClause* examples

The following example illustrates the translation of XSPARQL *SQLForClause*s into XSPARQL relational algebra expressions.

Example 4.7 (Translation of *SQLForClause*s into Relational Algebra)**.** Figure 4.4 presents two XSPARQL queries including *SQLForClause*s. The query in Figure 4.4a illustrates the syntax for querying values of a relation. First the normalisation function drops the restriction in line 3, which is incorporated into the relational algebra σ expression:

$$\sigma_{band.bandId=xir_C.x}(\sigma_{band.bandName='Nightwish'}(band) \times xir_C) \quad ,$$

where $sort(xir_C) = \{ x \}$ and $xir_C(x) = 1$.

On the other hand, the query in Figure 4.4b shows how to match attribute names. The query in this figure is converted into the following relational algebra expression:

$$\sigma_{band.bandName='Nightwish'}(\sigma_{band.bandId=1}(band) \times xir_C) \quad ,$$

where $sort(xir_C) = \{ x \}$ and $xir_C(x) =$ 'bandName'.

Querying RDF Data

For querying RDF data, we extend the notion of SPARQL BGP (Definition 3.10) in order to provide SPARQL with the variable bindings from XQuery. For this we interpret the XQuery varValue dynamic environment component as a set of bindings in the spirit of SPARQL solution mappings (as presented in Definition 3.6). Along these lines, we will regard the varValue component of the dynamic environment in which a SPARQL graph pattern P is executed as the basis for the *XSPARQL instance mapping* of P. The transformation from the dynEnv.varValue into the XSPARQL instance mapping is defined next:

Definition 4.9 (XSPARQL instance mapping)**.** *Let C be an expression context, and furthermore let $D_C = dynEnv(C).varValue$ and $T_C = \text{statEnv}(C).varType$ the varValue component of the dynamic environment of C and be the varType component of the static environment of C, respectively. The* XSPARQL instance mapping *μ_C is a solution mapping where, for each mapping $v_i \to x_i \in D_C$, x_i is converted into an instance of type* `RDFTerm` *or an RDF* Collection *according to the following conditions:*

- *if $D_C(v_i) = ()$ and $T_C(v_i) =$* `RDFTerm` *or $T_C(v_i) =$* `SQLTerm` *then $\mu_C(D_C(v_i))$ is undefined;*
- *if $D_C(v_i) = ()$ and $T_C(v_i) \neq$* `RDFTerm` *and $T_C(v_i) \neq$* `SQLTerm` *then $\mu_C(D_C(v_i)) = ()$ is an empty RDF Collection;*
- *if $D_C(v_i)$ is a singleton sequence then $\mu_C(D_C(v_i)) = RDFTerm(D_C(v_i))$;*
- *if $D_C(v_i) = (e_1, \ldots, e_n)$, $n > 1$, is a sequence then $\mu_C(D_C(v_i)) = (RDFTerm(e_1) \cdots RDFTerm(e_n))$ to be read as an RDF Collection in Turtle notation (cf. Section 2.4.1).*

For a graph pattern P, we call the XSPARQL instance mapping of the expression context in which P is executed the XSPARQL instance mapping of P.

```
1 declare namespace mo = "http://purl.org/ontology/mo/" ;
2 declare namespace dc = "http://purl.org/dc/elements/1.1/" ;
3
4 for $song from <bands.ttl>
5 where { $song a mo:Track }
6 return
7   for $songName from <bands.ttl>
8   where { $song dc:title $songName }
9   return <songName>{$songName}</songName>
```

Query 4.6: Nested XSPARQL query

Next we define the notion of XSPARQL BGP matching based on the semantics of SPARQL BGP matching presented in Section 3.3.

Definition 4.10 (Extended solution mapping)**.** *Let C be an expression context. An* extended solution mapping *of a graph pattern P in C is a solution mapping* compatible *with the* XSPARQL instance mapping *of C.*

Accordingly, XSPARQL BGP matching is defined analogously to the SPARQL BGP matching with the exception that we consider only extended solution mappings:

Definition 4.11 (XSPARQL BGP matching)**.** *Let P be a BGP with expression context C, and G be an RDF graph. We say that μ is a* solution *for P with respect to active graph G, if there exists an extended solution mapping μ' of C such that $\mu'(P)$ is a subgraph of G and μ is the restriction of μ' to the variables in $vars(P)$.*

This definition quasi injects the variable bindings inherited from XQuery into SPARQL patterns occurring within XSPARQL. By considering *extended solution mappings* the bindings returned for a BGP P will not only match the input graph G but also respect any bindings for variables in the dynamic environment. We can extend the XSPARQL BGP matching to generic graph patterns by following the SPARQL evaluation semantics (presented in Section 3.3). Considering a graph pattern P with XSPARQL instance mapping μ_C, we denote by $eval_{xs}(D, P, \mu_C)$ the evaluation of P over dataset D following XSPARQL BGP matching.

Matching Blank Nodes in Nested Queries

Although in XSPARQL, similar to SPARQL, we are not considering blank nodes in the semantics definitions of graph patterns, in the case of nested *SparqlForClause*s XSPARQL instance mappings may in fact contain assignments of variables to blank nodes, injected from the outer *SparqlForClause* into the inner *SparqlForClause*.

Example 4.8 (Blank node injection in XSPARQL nested queries)**.** For example, in Query 4.6, blank nodes bound in the outer *SparqlForClause* (lines 4–5) to the variable $*song* will be injected into the inner *SparqlForClause* expression (lines 7–8). If we would consider both *SparqlForClause*s as distinct SPARQL queries, the blank nodes in the inner *SparqlForClause* would be matched as variables.

However in XSPARQL, we want to enable coreference within nested queries over the same dataset and thus such injected blank nodes should be matched like constants against the blank nodes present in the input RDF data (rather than being treated as variables). To ensure this behaviour, we introduce the notion of *active dataset* (similar to the concept of active graph in SPARQL), where nested queries over the same active dataset keep the same the scoping graphs (cf. Section 3.3). Any *SparqlForClause* with

an *explicit DatasetClause* causes the *active dataset* to change, i.e. new scoping graphs (with fresh blank nodes) for each graph within it are created. On the other hand, if no *DatasetClause* is present in a nested *SparqlForClause* (implicit dataset), the active dataset remains unchanged. To ensure this behaviour in the dynamic evaluation we have to introduce a new dynamic environment component called activeDataset, that will be used to evaluate *WhereClause*s. Initially, this component is empty (or set to a system default) and is changed by a *DatasetClause* appearing in a *SparqlForClause*, as defined in the next section.

4.2.3. Extensions to the XQuery Semantics

In order to define the XSPARQL semantics according to XQuery's semantics we need to introduce new environment components and extend the dynamic evaluation rules of XQuery *ForClause*s to populate these new components. We also introduce the functions that we will use in the dynamic evaluation rules presented in the next section.

New Environment Components

For the definition of the XSPARQL semantics we add the following components to the dynamic environment:

(i) activeDataset; and
(i) globalPosition.

The dynEnv.activeDataset is used to store the dataset over which *SparqlForClause*s are evaluated in order to be accessible when a nested *SparqlForClause* without a *DatasetClause* is specified.

The other introduced environment component, dynEnv.globalPosition, stores all the positions in the tuple streams. The standard XQuery dynamic evaluation rules can only access the position of the current tuple stream, however, in order to generate distinct blank node labels for each *ConstructClause*, we need to guarantee that the labels are also distinct in case of nested queries. To ensure this, we store not only the position in the current tuple stream but also the positions of all previous ones.

Both environment components are populated in the dynamic evaluation rules introduced in Section 4.2.4. For the dynEnv.globalPosition we also need to adapt the evaluation rules of XQuery *ForClause*s to correctly populate this component. These updated rules are presented next.

XQuery `for` Dynamic Evaluation

In order to correctly generate blank node identifiers in *ConstructClause*s, XSPARQL relies on the dynEnv.globalPosition environment component to store the positions. As such, we adapt the XQuery `for` dynamic evaluation rules, presented in Section 3.2.3, to populate the dynEnv.globalPosition component and also make sure that the newly introduced XSPARQL *SQLForClause*s and *SparqlForClause*s populate this component. The case of these newly XSPARQL expressions is detailed later in Section 4.2.4.

We show here only the adapted rule for *ForClause*s with position variables and without type declaration. The rules that handle `for` expressions without position variables and possibly containing type declarations are adapted analogously, adding the dynEnv.globalPosition premisses to the rules presented in Draper,

Fankhauser et al. (2010, Section 4.8.2).

$$\frac{\begin{array}{c}\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_m) \\ \text{dynEnv} \vdash Expr_1 \Rightarrow Item_1, \cdots, Item_n \\ \text{statEnv} \vdash VarName \textbf{ of var expands to } Variable \\ \text{statEnv} \vdash VarName_{pos} \textbf{ of var expands to } Variable_{pos} \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_m, 1)) \\ \quad + \text{varValue}\left(\begin{array}{l} Variable \Rightarrow Item_1; \\ Variable_{pos} \Rightarrow 1 \end{array}\right) \quad \vdash Expr_2 \Rightarrow Value_1 \\ \vdots \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_m, n)) \\ \quad + \text{varValue}\left(\begin{array}{l} Variable \Rightarrow Item_n; \\ Variable_{pos} \Rightarrow n \end{array}\right) \quad \vdash Expr_2 \Rightarrow Value_n \end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$VarName \texttt{ at } \$VarName_{pos} \texttt{ in } Expr_1 \\ \texttt{return } Expr_2\end{array} \Rightarrow Value_1, \cdots, Value_n} \tag{D3}$$

New Formal Semantics Functions

Next we will introduce the new XQuery formal semantics functions that we use in the static and dynamic evaluation rules presented in the next section. These functions are specified in an informal style, in a similar fashion to formal semantics functions presented in Draper, Fankhauser et al. (2010, Section 7.1) and the XQuery 1.0 and XPath 2.0 Functions and Operators specifications (Malhotra et al., 2010). For each function, we present its signature, consisting of the function name, the function parameters, and the return type, and include a textual description of the semantics of the function.

The first introduced functions, *fs*:*sql* and *fs*:*sparql*, represent the extended SQL and SPARQL querying facilities implemented in XSPARQL (described in Section 4.2.2). We further introduce two auxiliary functions *fs*:*value*, *fs*:*dataset*, and *fs*:*evalCT*. These functions are used to access the value of a variable in a `PatternSolution`, to determine the dataset over which a *SparqlForClause* is evaluated, and to evaluate a `construct` query, i.e. a *ConstructTemplate*, respectively.

fs:sql This function is responsible for executing the extended XSPARQL SQL querying presented in Section 4.2.2. In our semantics this function also implements the normalisation of *SQLWhereClauses* (presented in Definition 4.6) by receiving two parameters: *RelationList* and *SQLWhereClause* representing the list of relations involved in the query and the SQL *pattern* to be executed, respectively. The static type signature of this function is defined as:

```
fs:sql($SparqlWhere as xs:string)
  as PatternSolution*
```

The replacement of variables in `SQLWhereClauses` represented by Definition 4.6 (that this function implements), produces a valid SQL query, that can be evaluated directly by the relational engine. The results of this query are then translated into an instance of `PatternSolution` (according to Definition 4.2).

fs:sparql. The *fs*:*sparql* function corresponds to the implementation of the $eval_{xs}$ function, that evaluates SPARQL graph patterns and implements the extended notion of BGP Matching presented in Definition 4.11. The static type signature of this function is defined as:

```
fs:sparql($dataset as RDFDataset, $SparqlWhere as xs:string, $solutionModifiers as xs:string)
  as PatternSolution*
```

The result of this function consists of a solution sequence, which can be translated directly into an XQuery sequence of elements of type `PatternSolution` by applying the *serialise* function (cf. Definition 4.1).

fs:value. The *fs:value*(PS, var) function returns the value of the specified variable *var* in a `PatternSolution` specified by PS. If *var* is not present in PS, the empty sequence is returned. The static type signature of this function is:

```
fs:value($ps as PatternSolution, $variable as xs:string)
  as (RDFTerm | SQLTerm)?
```

This function returns the respective `Binding` for the variable, which is an element of type `SQLTerm` or `RDFTerm`, depending on whether the pattern solution was the result of a *SQLForClause* or a *SparqlForClause*.

fs:dataset. The *fs:dataset*(*DatasetClause*) auxiliary function returns an element of type `RDFDataset` based on the evaluation of its argument. This conversion is performed according to the SPARQL semantics presented in Section 3.3. The result of this function is stored (by dynamic evaluation rules) in the newly introduced activeDataset dynamic environment component and can be retrieved when a *SparqlForClause* without an explicit *DatasetClause* is found. The static type signature of this function is:

```
fs:dataset($datasetClause as xs:string)
  as RDFDataset
```

fs:evalCT. The *fs:evalCT* function ensures the created RDF graph is valid and rewrites any blank nodes inside of *ConstructTemplate*s to comply with the SPARQL semantics (as described in Section 4.2.2). The auxiliary *fs:validTriple* function checks if each triple is valid according to the RDF semantics and is defined by rules (D10) and (D11) presented in the next section. The *fs:evalCT* function is further detailed in the following section by presenting specific rules that ensure the generated RDF graph is valid and to guarantee the generation of new blank node labels for each pattern solution. The static type signatures of these functions are defined as:

```
fs:evalCT($template as RDFTerm*)
  as RDFGraph

fs:validTriple($subject as RDFTerm, $predicate as RDFTerm,  $object as RDFTerm)
  as RDFTriple?
```

The *fs:evalCT* function, and hence `construct` expressions, return elements of type `RDFGraph`, thus allowing the result of `construct` expressions to be used in a *DatasetClause* of a subsequent *SparqlForClause*.

4.2.4. Semantics Rules for XSPARQL Expressions

We are now ready to present the normalisation, static, and dynamic evaluation rules for the newly defined XSPARQL expressions. As presented in Section 4.1, XQuery and SPARQL namespace declarations can be used interchangeably in the prolog of an XSPARQL query and thus we start by presenting the translation of the query prolog into XQuery namespace declarations via normalisations rules. We then present the necessary normalisation, static, and dynamic evaluation rules for *SQLForClause*s, *SparqlForClause*s, and *ConstructClause*s. Please note that, since the variables included in *SQLForClause*s and *SparqlForClause*s are not allowed to contain a namespace prefix, we omit the rules handling the namespace expansion for the respective variables.

Query Prolog Normalisation

In order to follow the XQuery semantics, we convert any SPARQL syntax prefix declaration into XQuery namespace declarations by the following normalisation rules:

$$\begin{array}{c} [\![\texttt{prefix } \mathit{NCName}\texttt{: <}\mathit{IRI}\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare namespace } \mathit{NCName} \texttt{ = "}\mathit{IRI}\texttt{" ;}]\!]_{Expr} \end{array} \quad \text{(N2)}$$

The empty `prefix` declaration is converted into the default namespace for XML elements:

$$\begin{array}{c} [\![\texttt{prefix : <}\mathit{IRI}\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare default element namespace = "}\mathit{IRI}\texttt{" ;}]\!]_{Expr} \end{array} \tag{N3}$$

Furthermore the SPARQL `base` declaration is considered equivalent to the XQuery `base-uri` declaration:

$$\begin{array}{c} [\![\texttt{base <}\mathit{IRI}\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare base-uri "}\mathit{IRI}\texttt{" ;}]\!]_{Expr} \end{array} \tag{N4}$$

SQLForClause

In this section we define the semantics of the newly introduced *SQLForClause* by means of the normalisation rules, static type analysis rules, and dynamic evaluation rules.

Normalisation rules. Let us start by presenting the normalisation rule that handles the syntactic shortcut '`for *`'.

$$\begin{array}{c} [\![\texttt{for} * \mathit{RelationList}\ \mathit{SQLWhereClause}\ \mathit{ReturnClause}]\!]_{Expr} \\ == \\ \left[\!\!\left[\begin{array}{l} \texttt{for}\ [\![\mathit{RelationList}\ \mathit{SQLWhereClause}]\!]_{attrs}\ \mathit{RelationList} \\ \mathit{SQLWhereClause}\ \mathit{ReturnClause} \end{array} \right]\!\!\right]_{Expr} \end{array} \tag{N5}$$

The normalisation rule $[\![\cdot]\!]_{attrs}$ returns a comma separated list of variables representing all the attributes from each relation from *RelationList*. As described in Section 4.1.3, these generated variables are of the form: $\$\mathit{relationName.attributeName}$. Furthermore, the next normalisation rule guarantees that each variable in a *SQLForClause* contains a variable alias:

$$\begin{array}{c} \left[\!\!\left[\begin{array}{l} \texttt{for}\ \mathit{AttrSpec}_1, \cdots, \mathit{AttrSpec}_n \\ \mathit{RelationList}\ \mathit{SQLWhereClause} \\ \mathit{ReturnClause} \end{array} \right]\!\!\right]_{Expr} \\ == \\ \begin{array}{l} \texttt{for}\ [\![\mathit{AttrSpec}_1]\!]_{Alias}, \cdots, [\![\mathit{AttrSpec}_n]\!]_{Alias} \\ \mathit{RelationList}\ \mathit{SQLWhereClause} \\ [\![\mathit{ReturnClause}]\!]_{Expr} \end{array} \end{array} \tag{N6}$$

A new normalisation rule $[\![\cdot]\!]_{Alias}$ takes care of introducing the variable alias when necessary, where the variable alias will be the same as the attribute specification.

$$[\![\mathit{AttrSpec}]\!]_{Alias} == \mathit{AttrSpec}\ \texttt{as}\ \$\mathit{AttrSpec}\ .$$

In case a variable alias is already present it is reused:

$$[\![\mathit{AttrSpec}\ \texttt{as}\ \$\mathit{VarRef}]\!]_{Alias} == \mathit{AttrSpec}\ \texttt{as}\ \$\mathit{VarRef}\ .$$

Static type analysis. The following static type rule defines the type of each variable in an *SQLForClause* as `SQLTerm` and infers the static type of whole expression. This rule, based on the static environment statEnv, creates a new environment with the added information that each of the variables in the *SQLForClause* ($\$\mathit{Var}_1 \ldots \$\mathit{Var}_n$) is of type `xs:anySimpleType`. Given this new extended environment the type of *ReturnExpr* can be inferred to be *Type*, making the type of the overall *SQLForClause* a

sequence of elements of inferred type *Type*.

$$\frac{\text{statEnv} + \text{varType}\left(\begin{array}{l} Var_1 \Rightarrow \texttt{SQLTerm}; \\ \ldots; \\ Var_n \Rightarrow \texttt{SQLTerm} \end{array}\right) \vdash ReturnExpr\colon Type}{\text{statEnv} \vdash \begin{array}{l} \texttt{for } AttrSpec_1 \texttt{ as } \$Var_1, \cdots, AttrSpec_n \texttt{ as } \$Var_n \\ RelationList\ SQLWhereClause \texttt{ return } ReturnExpr \end{array} : Type*} \tag{S2}$$

Dynamic Evaluation. The dynamic evaluation rules for *SQLForClause*s ensures that the return expression (*ReturnExpr*) is executed for each `SQLResult` that is returned by the evaluation of the SQL expression. If the evaluation of the SQL expression does not yield any solutions, i.e. evaluates to an empty sequence, then the overall result will also be the empty sequence:

$$\frac{\text{dynEnv} \vdash fs{:}sql(RelationList, SQLWhereClause) \Rightarrow ()}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } \$Var_1\ OptVarAlias_1, \ldots, \$Var_n\ OptVarAlias_n \\ RelationList\ SQLWhereClause \texttt{ return } ReturnExpr \end{array} \Rightarrow ()} \tag{D4}$$

Otherwise, for each solution, the respective value in the pattern solution is accessed and assigned to the respective variable name in the dynEnv.varValue component. The results of evaluating *ReturnExpr* in this extended environment are then collected into the final sequence. Please note that this rule also populates the dynEnv.globalPosition environment.

$$\frac{\begin{array}{r} \text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_j) \\ \text{dynEnv} \vdash fs{:}sql(RelationList, SQLWhereClause) \Rightarrow SR_1, \ldots, SR_m \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_j, 1)) \\ + \text{varValue}\left(\begin{array}{l} Var_1 \Rightarrow fs{:}value(SR_1, Var_1); \\ \ldots; \\ Var_n \Rightarrow fs{:}value(SR_1, Var_n) \end{array}\right) \vdash ReturnExpr \Rightarrow Value_1 \\ \vdots \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_j, m)) \\ + \text{varValue}\left(\begin{array}{l} Var_1 \Rightarrow fs{:}value(SR_m, Var_1); \\ \ldots; \\ Var_n \Rightarrow fs{:}value(SR_m, Var_n) \end{array}\right) \vdash ReturnExpr \Rightarrow Value_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } AttrSpec_1 \texttt{ as } \$Var_1 \ldots AttrSpec_n \texttt{ as } \$Var_n \\ RelationList\ SQLWhereClause \texttt{ return } ReturnExpr \end{array} \Rightarrow Value_1, \ldots, Value_m} \tag{D5}$$

SparqlForClause

The semantics of the *SparqlForClause* expression (Figure 4.2) is defined by the following normalisation rules, static type analysis rules and dynamic evaluation rules. Again, we start by presenting the normalisation rules for *SparqlForClause*s with implicit variable selection (by means of "`for *`"), which are translated into explicitly stated variables:

$$\begin{array}{c} \left[\!\left[\begin{array}{l} \texttt{for *}\ OptDatasetClause\ WhereClause \\ SolutionModifier \texttt{ return } ExprSingle \end{array} \right]\!\right]_{Expr} \\ == \\ \left[\!\left[\begin{array}{l} \texttt{for } [\![WhereClause]\!]_{vars} \\ OptDatasetClause\ WhereClause \\ SolutionModifier \texttt{ return } ExprSingle \end{array} \right]\!\right]_{Expr} \end{array} \tag{N7}$$

The normalisation rule $[\![WhereClause]\!]_{vars}$ determines all statically *unbound variables* present in the *WhereClause*, i.e. returns a whitespace separated list of all variables in the *WhereClause* that are not

present in the statEnv.varType environment component.

Static type analysis. The following static rule takes care of defining the types of variables present in a `for` expression as `RDFTerm` and infers the static type of the *SparqlForClause* expression:[2]

$$\frac{\text{statEnv} + \text{varType}\left(\begin{array}{l} \mathit{Var}_1 \Rightarrow \texttt{RDFTerm}; \\ \cdots ; \\ \mathit{Var}_n \Rightarrow \texttt{RDFTerm} \end{array}\right) \vdash \mathit{ExprSingle}\colon \mathit{Type}}{\text{statEnv} \vdash \begin{array}{l} \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \ \mathit{OptDatasetClause} \\ \mathit{WhereClause}\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle}\colon \mathit{Type}* \end{array}} \tag{S3}$$

Dynamic Evaluation. We can now define the dynamic evaluation rules for the *SparqlForClause* expression. Intuitively these rules state that the return expression *ExprSingle* will be executed for each `PatternSolution` that is returned from the evaluation of the *fs*:*sparql* function. The following two dynamic rules specify the evaluation of the *SparqlForClause* with an explicit *DatasetClause*. These rules use the *fs*:*dataset* function to parse the *DatasetClause* into an element of type `RDFDataset`, which will be stored in the dynEnv.activeDataset component: If the evaluation of the *fs*:*sparql* function does not yield any solutions, i.e. evaluates to an empty sequence, the overall result will also be the empty sequence:

$$\frac{\begin{array}{c} \text{dynEnv} \vdash \mathit{fs}\text{:}\mathit{dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs}\text{:}\mathit{sparql}\left(\begin{array}{l} \mathit{Dataset}, \mathit{WhereClause}, \\ \mathit{SolutionModifier} \end{array}\right) \Rightarrow () \end{array}}{\text{dynEnv} \vdash \begin{array}{ll} \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \ \mathit{DatasetClause} & \\ \mathit{WhereClause}\ \mathit{SolutionModifier} & \Rightarrow () \\ \texttt{return}\ \mathit{ExprSingle} & \end{array}} \tag{D6}$$

Otherwise, *ExprSingle* is evaluated for each solution in the results of the SPARQL query:

$$\frac{\begin{array}{c} \text{dynEnv.globalPosition} = (\mathit{Pos}_1, \cdots, \mathit{Pos}_j) \\ \text{dynEnv} \vdash \mathit{fs}\text{:}\mathit{dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs}\text{:}\mathit{sparql}\left(\begin{array}{l} \mathit{Dataset}, \mathit{WhereClause}, \\ \mathit{SolutionModifier} \end{array}\right) \Rightarrow \mu_1, \ldots, \mu_m \\ \text{dynEnv} + \text{globalPosition}((\mathit{Pos}_1, \cdots, \mathit{Pos}_j, 1)) \ + \text{activeDataset}(\mathit{Dataset}) \\ + \text{varValue}\left(\begin{array}{l} \mathit{Var}_1 \Rightarrow \mathit{fs}\text{:}\mathit{value}(\mu_1, \mathit{Var}_1)\,; \\ \ldots\,; \\ \mathit{Var}_n \Rightarrow \mathit{fs}\text{:}\mathit{value}(\mu_1, \mathit{Var}_n) \end{array}\right) \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_1 \\ \vdots \\ \text{dynEnv} + \text{globalPosition}((\mathit{Pos}_1, \cdots, \mathit{Pos}_j, m)) \ + \text{activeDataset}(\mathit{Dataset}) \\ + \text{varValue}\left(\begin{array}{l} \mathit{Var}_1 \Rightarrow \mathit{fs}\text{:}\mathit{value}(\mu_m, \mathit{Var}_1)\,; \\ \ldots\,; \\ \mathit{Var}_n \Rightarrow \mathit{fs}\text{:}\mathit{value}(\mu_m, \mathit{Var}_n) \end{array}\right) \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{ll} \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \ \mathit{DatasetClause} & \\ \mathit{WhereClause}\ \mathit{SolutionModifier} & \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \\ \texttt{return}\ \mathit{ExprSingle} & \end{array}} \tag{D7}$$

This rule ensures that the activeDataset component of the dynamic environment is updated to reflect the explicit *DatasetClause* of the *SparqlForClause* and that the globalPosition environment contains all the positions in the previous tuple streams.

[2] Similar to the XQuery Core *OptPositionalVar*, the *OptDatasetClause* covers both cases when a *SparqlForClause* contains (or does not contain) a *DatasetClause*.

The rule that handles the *SparqlForClause* without an explicit *DatasetClause* is presented next. These rules are very similar, with the exception that in following rules, the dataset over which the *SparqlForClause* is evaluated is read from the dynEnv.activeDataset component.

$$\frac{\begin{array}{c}\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_j) \\ \text{dynEnv.activeDataset} \Rightarrow Dataset \\ \text{dynEnv} \vdash fs{:}sparql\left(\begin{array}{l} Dataset, WhereClause, \\ SolutionModifier \end{array}\right) \Rightarrow \mu_1, \ldots, \mu_m \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_j, 1)) \\ \quad + \text{varValue}\left(\begin{array}{l} Var_1 \Rightarrow fs{:}value(\mu_1, Var_1)\,; \\ \ldots\,; \\ Var_n \Rightarrow fs{:}value(\mu_1, Var_n) \end{array}\right) \vdash ExprSingle \Rightarrow Value_1 \\ \vdots \\ \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_j, m)) \\ \quad + \text{varValue}\left(\begin{array}{l} Var_1 \Rightarrow fs{:}value(\mu_m, Var_1)\,; \\ \ldots\,; \\ Var_n \Rightarrow fs{:}value(\mu_m, Var_n) \end{array}\right) \vdash ExprSingle \Rightarrow Value_m \end{array}}{\begin{array}{l} \texttt{for}\ \$Var_1 \cdots \$Var_n \\ \text{dynEnv} \vdash \ WhereClause\ SolutionModifier\ \Rightarrow Value_1, \ldots, Value_m \\ \texttt{return}\ ExprSingle \end{array}} \tag{D8}$$

Analogously to the *SparqlForClause* with an explicit dataset (Rule D6), whenever the *fs:sparql* function evaluates to an empty sequence, the result will also be an empty sequence.

ConstructClause

XSPARQL normalises *ConstructClause*s into standard XQuery `return` expressions with the necessary mechanisms for validation of the returned RDF graph and as such, we define the semantics of *ConstructClause*s (Figure 4.2) by means of normalisation rules. One valid syntax for XSPARQL is a SPARQL stand-alone `construct` query (as described in Section 4.1). These queries are normalised into `construct` queries with a surrounding *SparqlForClause* by the following rule:

$$\begin{array}{c} \left[\!\!\left[\begin{array}{l} \texttt{construct}\ ConstructTemplate \\ DatasetClause\ WhereClause \\ SolutionModifier \end{array} \right]\!\!\right]_{Expr} \\ == \\ \left[\!\!\left[\begin{array}{l} \texttt{for}\ *\ DatasetClause \\ WhereClause\ SolutionModifier \\ \texttt{construct}\ ConstructTemplate \end{array} \right]\!\!\right]_{Expr} \end{array} \tag{N8}$$

The recursive call to $[\![\cdot]\!]_{Expr}$ ensures that the resulting query will be further rewritten according to normalisation Rule (N7) presented above, in order to explicitly state the variables present in the *WhereClause*.

Similar to the normalisation rule for stand-alone *ReturnClauses* presented in Draper, Fankhauser et al. (2010, Section 4.8.1), the following normalisation rule transforms `construct` clauses into XQuery *ReturnClause*s.

$$\begin{array}{c} [\![\texttt{construct}\ ConstructTemplate]\!]_{Expr} \\ == \\ \texttt{return}\ fs{:}evalCT\left([\![ConstructTemplate]\!]_{normCT}\right) \end{array} \tag{N9}$$

In the following we assume that *ConstructTemplate* is a simple '.' separated list of *Subject*, *Predicate* and *Object*. The $[\![\cdot]\!]_{normCT}$ rule transforms any Turtle shortcut notation used in *ConstructTemplate* to

these simple lists. As an example of this rule, we present the rule for normalising Turtle '`;`' abbreviations (previously described in Section 2.4.1):

$$\begin{array}{c} [\![\mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1 ;\ \cdots ;\ \mathit{Pred}_n\ \mathit{Obj}_n\ .]\!]_{normCT} \\ == \\ \mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1\ .\ \cdots\ \mathit{Subject}\ \mathit{Pred}_n\ \mathit{Obj}_n\ . \end{array} \tag{N10}$$

The normalisation rules for the other Turtle shortcuts that are allowed in the SPARQL *ConstructTemplate* syntax are similar to this one and are not presented here.

Since anonymous blank nodes can be written in numerous ways in Turtle, the $[\![\cdot]\!]_{normCT}$ normalisation rule also transforms each anonymous blank node into a labelled blank node where the identifier/label is distinct from any other blank node labels present in the *ConstructTemplate*. This label will then be used by the skolemisation function to generate the distinct blank node label for each position in the tuple stream.

In more detail, the *fs*:*evalCT* function checks the constructed RDF graph for validity (according to the conditions described in Section 3.3), filtering out any non-valid RDF triples where *subjects* are literals or *predicates* are literals or blank nodes. This is illustrated by the following dynamic evaluation rules.

$$\frac{\begin{array}{c} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{validTriple}(\mathit{Subj}_1, \mathit{Pred}_1, \mathit{Obj}_1) \Rightarrow \mathit{Triple}_1 \\ \vdots \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{validTriple}(\mathit{Subj}_n, \mathit{Pred}_n, \mathit{Obj}_n) \Rightarrow \mathit{Triple}_n \end{array}}{\text{dynEnv} \vdash \mathit{fs}{:}\mathit{evalCT} \begin{pmatrix} \mathit{Subj}_1\ \mathit{Pred}_1\ \mathit{Obj}_1 \\ \ldots \\ \mathit{Subj}_n\ \mathit{Pred}_n\ \mathit{Obj}_n \end{pmatrix} \Rightarrow \mathit{Triple}_1,\ \cdots,\ \mathit{Triple}_n} \tag{D9}$$

The following dynamic evaluation rule for the *fs*:*validTriple* function checks, relying on the *fs*:*bnode* function defined below, if a triple is valid according to the RDF semantics.

$$\frac{\begin{array}{c} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{bnode}(\mathit{Subject}) \Rightarrow \mathit{ValS} \\ \text{statEnv} \vdash \mathit{ValS}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode}) \\ \text{dynEnv} \vdash \mathit{Predicate} \Rightarrow \mathit{ValP} \\ \text{statEnv} \vdash \mathit{ValP}\ \textbf{matches}\ \texttt{uri} \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{bnode}(\mathit{Object}) \Rightarrow \mathit{ValO} \\ \text{dynEnv} \vdash \mathit{ValO}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode} \mid \texttt{literal}) \end{array}}{\text{dynEnv} \vdash \mathit{fs}{:}\mathit{validTriple} \begin{pmatrix} \mathit{Subject}, \\ \mathit{Predicate}, \\ \mathit{Object} \end{pmatrix} \Rightarrow \begin{array}{l} \texttt{element triple of type RDFTriple } \{ \\ \quad \texttt{element subject of type RDFTerm } \{\mathit{ValS}\} \\ \quad \texttt{element predicate of type RDFTerm } \{\mathit{ValP}\} \\ \quad \texttt{element object of type RDFTerm } \{\mathit{ValO}\} \\ \} \end{array}} \tag{D10}$$

In case any of the subject, predicate or object do not match an allowed type, the empty sequence is returned. Effectively this suppresses any invalid RDF triples from the output graph.

$$\frac{\begin{array}{c} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{bnode}(\mathit{Subject}) \Rightarrow \mathit{ValueS} \\ \text{dynEnv} \vdash \mathit{Predicate} \Rightarrow \mathit{ValueP} \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{bnode}(\mathit{Object}) \Rightarrow \mathit{ValueO} \\ \text{dynEnv} \vdash \textbf{not} \begin{pmatrix} \mathit{ValueS}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode})\ \textbf{and} \\ \mathit{ValueP}\ \textbf{matches}\ \texttt{uri}\ \textbf{and} \\ \mathit{ValueO}\ \textbf{matches}\ (\ \texttt{uri} \mid \texttt{bnode} \mid \texttt{literal}\ ) \end{pmatrix} \end{array}}{\text{dynEnv} \vdash \mathit{fs}{:}\mathit{validTriple}(\mathit{Subject}, \mathit{Predicate}, \mathit{Object}) \Rightarrow ()} \tag{D11}$$

Blank Node Skolemisation. In order to comply with the SPARQL `construct` semantics, all blank nodes inside a *ConstructTemplate* need to be *skolemised*, i.e. for each solution a new distinct blank node identifier needs to be generated. Since we keep all the positions in the tuple streams, we can rely on

the blank node label and these position values to generate a unique blank node label (represented by the *fs:skolemConstant* function). This skolemisation of blank nodes is performed by the *fs:bnode* function. If the argument of this function is of type `bnode` the skolemised label is calculated.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{ValueR}\ \textbf{matches}\ \texttt{bnode} \\ \text{dynEnv.globalPosition} = (\mathit{PosValue}_1, \cdots, \mathit{PosValue}_n) \\ \text{dynEnv} \vdash \mathit{fs{:}skolemConstant}\left(\begin{array}{l}\mathit{ValueR}, \\ \mathit{PosValue}_1, \\ \ldots, \\ \mathit{PosValue}_n\end{array}\right) \Rightarrow \mathit{ValueRS}\end{array}}{\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{ValueR}) \Rightarrow \texttt{element bnode of type xs:string}\ \{\mathit{ValueRS}\}} \tag{D12}$$

Otherwise, *fs:bnode* returns its argument unchanged:

$$\frac{\text{dynEnv} \vdash \mathit{Value}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{literal})}{\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{Value}) \Rightarrow \mathit{Value}} \tag{D13}$$

4.3. Semantic Correspondence between XSPARQL, SQL, XQuery, and SPARQL

Since XSPARQL syntactically extends XQuery, and also any SPARQL `construct` query is syntactically valid in XSPARQL, these queries are considered semantically equivalent to the semantics in their base languages. Regarding SQL and SPARQL `select` expressions, we can show that their results remain the same under XSPARQL extended semantics. The next propositions formally establish this intuitive correspondence.

Proposition 4.1. *XSPARQL is a conservative extension of XQuery.*

Proof: The additional rules introduced in Section 4.2 do not modify the semantics of any native XQuery: the XSPARQL semantics – expressed in terms of normalisation rules, static typing rules and dynamic evaluation rules – strictly extends the native semantics of XQuery. In the semantics definition we also define new environment components, namely statEnv.globalPosition and dynEnv.activeDataset, which are not used in the XQuery semantics and thus do not interfere with query evaluation. The only rules that use these newly created environments are the evaluation rules of *SparqlForClause*s (dynEnv.activeDataset) and the dynamic evaluation rule (D12) (dynEnv.globalPosition), which generates Skolem-identifiers for blank nodes in `construct` parts. However, all these rules only apply to XSPARQL queries, which fall outside the native XQuery fragment, whereas the semantics of native XQuery queries remains untouched and independent of the extra environment components in XSPARQL. □

We can also show that the answers of an XSPARQL *SQLForClause* without any previously bound XSPARQL variables are the same as the answers of the normalised query under SQL semantics:

Lemma 4.2. *Let S be a SQLForClause, xir_C the XSPARQL instance relation of S, and $S' = normaliseSQL(S)$ the SQL normalised query of S. Furthermore, let $R_1 = RA_{xsp}(S)$ and $R_2 = RA_{sql}(S')$, where $sort(R_2) = U$ be the relation instances resulting from evaluating S according to the XSPARQL semantics and the SQL semantics, respectively. If S does not contain any XSPARQL variables, i.e. $vars(S) = \emptyset$, then $R_1[U] = R_2$.*

Proof: Following Definition 4.8 we have that the answers of S under XSPARQL semantics are given by $\sigma_E(R_1 \times xir_C)$, where E is the XSPARQL select expression of S. Since $vars(S) = \emptyset$, according to Definition 4.7, E will also be empty and we can simplify the expression that produces R_1 to $R_2 \times xir_C$. According to the definition of XSPARQL instance relation (Definition 4.5) xir_C has cardinality 1 and

thus the cross product does not change the cardinality of R_2, simply extending each solution in R_2 with the attributes from the xir_C relation. Since the cardinality of R_1 and R_2 is the same and the $\times$ operation does not change any existing attributes of R_1, we have that $R_1[U] = R_2$. □

Similarly for SPARQL, we show the equivalence between SPARQL BGP Matching (Prud'hommeaux and Seaborne, 2008, Section 12.3.1) and XSPARQL BGP Matching (presented in Section 4.2.2). Based on this, we can then prove the equivalence between the XSPARQL and SPARQL semantics for `construct` queries.

Lemma 4.3. *Given a graph pattern P, a dataset D and μ_C the XSPARQL instance mapping of P. Furthermore, let $\Omega_1 = eval_{xs}(D, P, \mu_C)$ and $\Omega_2 = eval(D, P)$ be solution mappings. If $vars(P) \cap dom(\mu_C) = \emptyset$, then $\Omega_1 = \Omega_2 \bowtie \{ \mu_C \}$.*

Proof: The XSPARQL BGP matching, $eval_{xs}(D, P, \mu_C)$, extends SPARQL's BGP matching, $eval(D, P)$, by defining that the solutions of the BGP are the ones *compatible* with the *XSPARQL instance mapping* μ_C. Since the evaluation of graph patterns (such as `union`, `optional`, `graph` and `filter`) remains unchanged from the SPARQL semantics let us focus on the evaluation of a BGP P. If there are no shared values between the graph pattern and the XSPARQL instance mapping, as is the case when $vars(P) \cap dom(\mu_C) = \emptyset$, then each solution $\mu \in \Omega_2$ returned by the SPARQL BGP evaluation semantics is trivially compatible with μ_C and the result of the XSPARQL BGP matching is $\mu \cup \mu_C$. Extending this result to all solution mappings in Ω_2, we obtain that $\Omega_1 = \Omega_2 \bowtie \{ \mu_C \}$. □

Finally, for SPARQL `construct` queries we can state the following:

Proposition 4.4. *XSPARQL is a conservative extension of SPARQL `construct` queries.*

Proof: For XSPARQL queries consisting of a standalone SPARQL `construct` query, there cannot exist any previous bindings for variables in XSPARQL and thus the XSPARQL instance mapping μ_C over which the `construct` query will be executed is empty. Let P represent the graph pattern of the `construct` query and D the dataset, since μ_C is empty, trivially there are no shared variables between μ_C and P. Thus, following Lemma 4.3 the bindings for XSPARQL BGP matching (say Ω_1) are the same bindings as SPARQL BGP matching (Ω_2), since $\Omega_1 = \Omega_2 \cup \{\emptyset\}$ and hence $\Omega_1 = \Omega_2$. Furthermore the formal semantics function *fs:evalTemplate* returns an RDF graph satisfying all the conditions of Definition 3.14: (i) ignoring invalid RDF triples – item (1) – is guaranteed by Rules (D10) and (D11); and (ii) the generation of distinct blank nodes for each solution sequence – item (2) – is enforced by the blank node skolemisation rules, Rules (D12) and (D13). □

4.4. Consuming JSON Data

Due to the similarity between JSON and XML, in XSPARQL we incorporate JSON data by translating the JSON objects into XML data. Furthermore JSON does not specify a query language (this representation format is meant to be incorporated directly into the JavaScript scripting language). As presented in Section 2.3, XML is more flexible than JSON and it is possible to convert JSON into XML but not so easy in the opposite direction.

This translation of JSON to XML enables access to the JSON data using standard XPath. The following definition presents the translation we use in XSPARQL.

Definition 4.12 (Translation from JSON to XML)**.** *Let J be a JSON object. The translation of J to XML, denoted translateXML(J), is an XML document* <jsonObject>*translateMembers(J)* </jsonObject>*, where translateMembers(J) is defined as follows:*

```
<jsonObject>
  <bands>
    <Nightwish>
      <albums>
        <Wishmaster>
          <arrayElement>Wishmaster</arrayElement>
          <arrayElement>FantasMic</arrayElement>
        </Wishmaster>
      </albums>
      <members>
        <arrayElement>Tuomas Holopainen</arrayElement>
        <arrayElement>Tarja Turunen</arrayElement>
      </members>
    </Nightwish>
  </bands>
</jsonObject>
```

Data 4.1: XML representation of JSON data

```
for $member in xsparql:json-doc("file:bands.json")//Nightwish/members/*
return data($member)
```

Query 4.7: Querying JSON using XSPARQL

- *if J is an empty JSON object or empty array, then* ();
- *if J is a JSON object, then for each $K_i : V_i \in J$,* <Ki>*translateMembers*(V_i)</Ki>;
- *if J is a JSON array, then for each $E_i \in J$,* <arrayElement>*translateMembers*(E_i)</arrayElement>;
- *otherwise J.*

For example the JSON from Data 2.2 translated into XML according to Definition 4.12 is presented in Data 4.1.

Querying the XML representation of JSON

JSON data can be manipulated directly in JavaScript, where accessing members of objects can be done using the '`.`' separator, while accessing array elements is done using the standard bracket notation: '`[`' and '`]`'. For example, if the JSON object in Data 2.2 is assigned to a JavaScript variable named '`b`', we can access the member '`bands`' by using '`b.bands`' and accessing the second member of the Nightwish band can be done with '`b.bands.Nightwish.members[1]`'.[3] In XSPARQL, querying the XML representation of JSON data can be done using an XPath expression, where, assuming *translateXML*(b) is assigned to an XSPARQL variable $\$b$:

(i) accessing members of an object can is done using the '`child`' XPath axis, for example to access the representation of member '`bands`' we write '$\$b$`/bands`'; and

(ii) accessing specific elements of an array can be done using XPath predicates, e.g. to access the second member of the Nightwish band can be done with '$\$b$`/bands/Nightwish/members/*[2]`'.[4]

[3] Please note that in JavaScript the first element of an array is at position 0, while the first element of XPath sequences is 1.
[4] We can also use '`arrayElement`' instead of '`*`' in the XPath expression.

Example 4.9 (Querying JSON using XSPARQL)**.** Query 4.7 presents the XSPARQL query that returns all members of the "`Nightwish`" band from the (translated) JSON Data 2.2. In an XSPARQL query the transformation from JSON into XML is implemented using the `xsparql:json-doc` function (as shown in line 1 of Query 4.7).

The implementation of the translation in Definition 4.12 currently translates the complete JSON provided as input. One possible optimisation for this implementation is to make it aware of the XPath expression and perform a selective translation of the input JSON data.

4.5. Processing RDB2RDF Mappings in XSPARQL

The W3C RDB2RDF Working Group (WG) is currently in the process of defining a standard language to translate a relational database into RDF. The WG has defined 2 documents: the *Direct Mapping (DM)* (Arenas, Prud'hommeaux et al., 2012) specifies the process of translating a relational database into RDF in an automated manner, and the *R2RML* language definition (Das, Sundara et al., 2012) corresponds to a user specified translation (in Turtle syntax) of the input relational database. The direct mapping provides a generic representation of the relational database while the R2RML provides more fine-tuned control over the produced RDF.

Next we start by giving an overview of the RDB2RDF Direct Mapping, the R2RML language, and then provide an algorithm for the implementation of R2RML in XSPARQL.

4.5.1. Direct Mapping

The aim of the DM is to provide an *off-the-shelf* translation of relational databases into RDF, i.e. a transformation that requires minimal user input. This translation follows already existing approaches, implemented by several conversion tools, and relies on creating the output RDF graph by assigning a unique identifier to each tuple in a relation from the input database. This identifier is created based on the relation name and the values for any existing primary keys and is then used as the subject of each RDF triple generated from the specific tuple.[5] Attributes names are used to generate a URI that is used as a predicate, while the object consists of the value for the specific attribute.

For processing DM in XSPARQL we need to have access to the underlying relational schema. For this we rely on a custom function that returns an XML representation of the relational schema and, based on this representation, the DM implementation is similar to the R2RML mappings, where we can use *SQLForClause*s to access the relational database and generate the target RDF graph. In the rest of this section we will focus on R2RML mappings and describe in more detail the XSPARQL query used to implement such transformations.

4.5.2. The R2RML mapping language

The R2RML mapping is itself an RDF graph consisting of several `TriplesMap`, that specify how to map a *logical table* in the input relational database into RDF. The *logical table* can correspond to a table, a view in the database, or the result of a SQL query to be executed over the input relational database.[6]

Each `TriplesMap` consists of one `SubjectMap` and possibly multiple `PredicateObjectMaps`. Each row in the logical table produces a single *subject* in the target RDF, which is specified by the `SubjectMap`. The

[5] In case a relation does have any primary keys a distinct blank node is used as an identifier for each tuple.
[6] Arbitrary SQL queries can be executed in XSPARQL via an implementation-defined XQuery function and were included only to cater for this feature of R2RML.

```
1  @prefix rr: <http://www.w3.org/ns/r2rml#> .
2  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
3  @prefix ex: <http://example.com/> .
4  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
5  @base <http://example.com/base/> .
6
7  <#TriplesMapBand> a rr:TriplesMap;
8    rr:logicalTable [ rr:tableName "band" ];
9    rr:subjectMap [ rr:template "http://example.com/band/{bandId}" ];
10
11   rr:predicateObjectMap [
12     rr:predicate foaf:name ; rr:objectMap [ rr:column "bandName" ] ];
13
14   rr:predicateObjectMap
15     [ rr:predicate foaf:member ;
16       rr:objectMap [ a rr:RefObjectMap ;
17                      rr:parentTriplesMap <#TriplesMapPerson>;
18                      rr:joinCondition [ rr:child "bandId" ; rr:parent "bandId" ; ] ] ] .
19
20
21 <#TriplesMapPerson> a rr:TriplesMap;
22   rr:logicalTable [ rr:tableName "person" ];
23   rr:subjectMap [ rr:template "http://example.com/person/{personId}" ];
24
25   rr:predicateObjectMap [
26     rr:predicate foaf:name ; rr:objectMap [ rr:column "personName" ] ] .
```

Figure 4.5.: RDB2RDF mapping for tables “band” and “person”

multiple `PredicateObjectMaps` each specify how to generate a *predicate* and possibly several *objects* (by means of `PredicateMaps` and `ObjectMaps`, respectively) that are related to the generated *subject*.

Furthermore, each `SubjectMap`, `PredicateMap`, and `ObjectMap` can specify how the RDF term is created by using different RDF predicates. For instance, using the `column` predicate for the mapping rule (e.g. the predicate of the `ObjectMap` on line 12 of Figure 4.5) indicates that the RDF object should be generated based on the value of the column in the input database. Another example is the `template` predicate, which specifies how terms are generated by using a template that will be instantiated with values from the logical table, e.g. the `subjectMap` from line 9 of Figure 4.5, states that the generated subject should be of the format

`http://example.com/band/{bandId}`

where `{bandId}` is to be replaced by the value of the “`bandId`” attribute in the specific tuple. The other predicate used in the example from Figure 4.5 is `rr:predicate` (line 15), which states that the predicate of the generated triples should be `foaf:name`.

Finally, foreign keys can be specified using `RefObjectMap` and by indicating the `TriplesMap` that represents the foreign logical table and possibly a `joinCondition` that specifies how to merge the two relations, as shown in lines 17–18 of Figure 4.5.

An R2RML mapping produces an RDF dataset with all the generated triples belonging to the default graph unless otherwise stated. To cater for the possibility of creating triples in a named graph, we extend XSPARQL’s generation of RDF graphs in Turtle format to generate an N-Quads representation (Cyganiak et al., 2009) of the RDF data. This extension is also used and is further expanded in Chapter 7.

Algorithm 1: *rdb2rdf*($\$m$)

Input: RDB2RDF mapping $\$m$ (represented as RDF)
Result: RDF Graph

```
1 let $mapSk := skolemise($m)
2 for * from $mapSk
3 where
4 |  $map rdf:type TriplesMap; rr:logicalTable $table; rr:subjectMap $s .
5 return
6 |  for row $tableRow in getLogicalTable($table) do
7 |  |  let $subject := createSubject($mapSk, $tableRow, $s)
8 |  |  createPO($mapSk, $tableRow, $subject, $map)
```

Algorithm 2: *createPO*($\$mapSk, \$row, \$subject, \$map$)

Input: skolemised RDB2RDF mapping $\$mapSk$, Database data $\$row$, generated RDF term $\$subject$, input RDF term $\$po$
Result: RDF Graph

```
1 for * from $mapSk
2 where
3 |  $map rr:predicateObjectMap [ rr:predicateMap $p; rr:objectMap $o ]
4 return
5 |  let $predicate := createTerm($mapSk, $row, $p)
6 |  let $object := createTerm($mapSk, $row, $o)
7 |  construct
8 |  |  $subject $predicate $object
```

4.5.3. R2RML Implementation in XSPARQL

In this section we present an algorithm that implements the R2RML transformation in XSPARQL. This transformation is implemented as an XSPARQL query but, for readability purposes, is summarised in Algorithm 1. In this algorithm we rely on multiple queries to the R2RML input mapping file and since the R2RML representation may use blank nodes for describing the mapping, we start by *skolemising* blank nodes in the input RDF graph, i.e. any blank nodes used in the R2RML mapping are substituted with newly generated URIs that are distinct from any other URI in the graph. This transformation allows us to use these newly generated URIs to merge data across different queries and is represented in the algorithm by the *skolemise* function (line 1).

The *SparqlForClause* on lines 2–8 iterates over all the `TriplesMaps` present in the mapping file and, for each of these `TriplesMaps`, retrieves the specified data from the input relational database. This access to the (logical) table of the relational database is represented by the *SQLForClause* on line 6, which instantiates $\$row$ for each result row that the corresponding SQL query returns (as described in Section 4.1). The function `getLogicalTable` is responsible for processing the different available forms of specifying the input relation in R2RML. In line 7 we generate the *subject* that is shared by all the triples derived from the same row of the relation and pass it to the `createPO` auxiliary function (line 8) that takes care of generating the predicate-object pairs.

The *createPO* function described in Algorithm 2 retrieves all the `predicateMap` and `objectMaps` associated with the `TriplesMap` we are processing (lines 1-3), creates the respective *predicate* (line 5) and *object* (line 6) and then generates an RDF triple using the XSPARQL built-in `construct` expression. The `construct` expression automatically takes care of discarding any non-valid RDF triples.

Algorithm 3: *createTerm*($*mapSk*, $*row*, $*spec*)

Input: skolemised RDB2RDF mapping $*mapSk*, Database data $*row*, RDF term specification $*spec*
Result: RDF Term

```
1 for * from $mapSk
2 where
3     $spec $specType $specValue
4 return
5     if $specType == rr:predicate then
6         createURI($specValue)
7     else if $specType == rr:column then
8         createLiteral(value($row, $specValue))
9     else ...
```

```
<http://example.com/band/1> <http://xmlns.com/foaf/0.1/name> "Nightwish" .
<http://example.com/band/1> <http://xmlns.com/foaf/0.1/member> <http://example.com/
    person/2> .
<http://example.com/band/1> <http://xmlns.com/foaf/0.1/member> <http://example.com/
    person/1> .
<http://example.com/person/2> <http://xmlns.com/foaf/0.1/name> "Tarja Turunen" .
<http://example.com/person/1> <http://xmlns.com/foaf/0.1/name> "Marco Hietala" .
```

Data 4.2: Output of algorithm rdb2rdf (Algorithm 1)

The auxiliary function `createTerm` is partially presented in Algorithm 3: given a specific database table *$row*, this function produces an RDF term according to the specification given in the RDB2RDF mapping. The *SparqlForClause* from lines 1-3 takes care of querying the RDB2RDF mapping to determine the type of the RDF term to be produced. Finally, the **return** clause (lines 4-9) presents the process of creating RDF terms for the `rr:predicate` and `rr:column` types of specifications. The `createURI` and `createLiteral` functions used in this algorithm are XSPARQL built-in functions that behave as constructors for URIs and literals, respectively. The `value` function returns the value associated with a column name in an `SQLResult` (similar to the formal semantics function presented in Section 4.2.3). The missing RDB2RDF specifications are similar to the presented ones possibly requiring some extra processing, e.g. the `rr:template` specification needs to be parsed to extract the column names from the template and then access their values of the current row.

Data 4.2 presents the RDF graph resulting from the applying Algorithm 1 to the RDB2RDF mapping presented in Figure 4.5.

4.6. Related Work

Several proposals for integrating data from relational databases, XML, and RDF were presented before. On one hand, converting between relational databases and XML has been long studied, either by the integration of SQL and XML (Eisenberg and Melton, 2001; Eisenberg and Melton, 2004) or the specification of the representation of database instances in XML.[7] In practice, most relational database management systems include a datatype for storing XML data while other works focus on the implementation of the XQuery language over a relational database backend (Grust, Sakr et al., 2004; Grust, Rittinger et al., 2008). As such, this section focuses on the integration of XML and RDF or relational databases and RDF data.

[7] `http://www.w3.org/XML/RDB.html`, retrieved on 2012/07/17.

Table 4.1.: Overview of Related Work

System	Input Format RDB	XML	RDF	Target Model	Query Language Surface	Target	Ontology Generation
Gloze		√		RDF	—	—	partial
Droop et al. (2008)		√		RDF	SPARQL+XSLT	SPARQL	×
Vrandecic et al. (2005)			√	—	—	SPARQL	×
Bohring and Auer (2005)		√		RDF	—	—	√
Rodrigues et al. (2008)		√		RDF	—	—	√
Fischer et al. (2011)	√	√	√	—	—	XQuery	×
Walsh (2003)		√	√	—	—	—	×
Berrueta et al. (2008)		√	√	RDF	XSLT+SPARQL	—	√
Bikakis et al. (2009)		√	√	—	SPARQL	XQuery	√
Groppe et al. (2008)		√	√	XML	SPARQL	XQuery	√
MarkLogic Server		√	√	XML	SPARQL	XQuery	√
Corby et al. (2009)	√	√	√	—	SPARQL	—	×
RDB2RDF	√		√	RDF	—	—	×
XSPARQL	√	√	√	XML/ RDF	XSPARQL	XQuery	×

To begin, an analysis of tools for converting between relational databases and RDF is presented by Gray et al. (2009), which also aims at studying the expressivity of SPARQL to represent scientific queries, namely in the astronomy domain. Although, as stated by the authors, data and queries were mostly numeric and thus biased towards relational data and SQL, the comparison gives a good overview of how the tested tools perform in comparison to relational databases. Some of the conclusions indicate that these tools are still not able to compete with relational databases in terms of performance and that SPARQL is also not yet expressive enough to pose the necessary queries.

Patel-Schneider and Siméon (2003) present a proposal to integrate the semantics of XML and RDF by defining a model-theory that encapsulates both the XDM and RDF data models. This proposal has not been applied in practice and most of the existing proposals to merge XML and RDF rely on translating the data from different formats and/or translating the queries from different languages. With this in mind, we divided the proposals into the following categories:

(1) **Normalised Representations:** include proposals that suggest using a normalised format for representing RDF in XML. Although similar to the next proposals, in these systems the translation can usually be automated and they do not address querying, simply reusing standardised languages.
(2) **Translation of data:** these tools aim at integrating the heterogeneous data by translating between different formats, usually relying on user predefined mappings.
(3) **Integration of query languages:** this category of approaches (where XSPARQL is also included) considers the integration and/or expansion of query languages to allow querying different formats without requiring the translation of data from the original formats.

Table 4.1 presents an overview of systems considered in (2) and (3). This table classifies the different systems according to whether they support input from relational databases (RDB), XML, or RDF. The target model indicates, if there exists a data translation step, what is the format used for the integrated representation. The Surface and Target query languages state, if available, the language in which the system accepts queries and if they are translated into a different query language. Finally, the Ontology Generation column specifies if the system generates an ontology description based an input XML Schema or relational database structure. We next give a short description of some of the tools and proposals available grouped by the presented categories.

Normalised Representations

The following proposals specify a normalised syntax for RDF in XML. The TriX format (Carroll and Stickler, 2004) consists of an alternative normalised serialisation for RDF in XML, with the aim of being compatible with standard XML tools. In this serialisation, each RDF triple is represented as a `triple` XML element with three children elements representing the subject, predicate, and object of the triple. It uses XSLT as an extensibility mechanism, allowing syntactic extensions to be specified and macros to be defined.

Also in 2003, *TreeHugger*[8] defines abstraction functions (implemented as extensions of the Saxon XQuery engine) that enable the navigation of an RDF graph structure in both XSLT and XQuery. This navigation is specified using XPath-like expressions that specify the RDF class and property that users want to query, which are in turn translated into SPARQL queries.

Well known parsers for RDF, such as the Redland RDF Libraries[9] also provide canonical formats of RDF/XML. *R3X*[10] takes this representation one step further by grouping the canonical RDF/XML output of the Redland parser and grouping the triples by subject. The aim of this grouping by subject is to make the canonical format easier to process with XSLT. Very similar to R3X, Grit[11] also defines a normalised format of RDF/XML where triples are grouped by their subject to facilitate processing in XSLT and improve the triple access evaluation times. Furthermore Grit normalises URIs to make lookups easier in XSLT.

Data Translation

We now present the proposals that rely on a user-specified normalised format for RDF. Gloze (Battle, 2006) aims at interpreting an XML document as RDF data based on the XML Schema definition. XML elements and attributes are mapped to RDF object or datatype properties, depending on wether they are described as complex or simple types in the XML Schema (complex types are mapped to object properties and simple types are mapped to datatype properties).

Droop et al. (2008) translate the XML document into RDF, annotating it with necessary information to answer XPath queries, namely the ordering, axes relations between XML elements, and attributes of elements. The authors then propose to integrate XPath queries into SPARQL as subqueries in BGPs, where the result of the subexpressions is assigned to SPARQL variables. These XPath subexpressions are in turn translated into SPARQL queries that, using the introduced annotations, allow the preservation of the semantics of the original queries and ordering of solutions.

Deursen et al. (2008) presents an approach for the transformation between XML and RDF by specifying mappings between an XML Schema and an OWL ontology. The authors introduce a language for the mapping specification, relying on XPath expressions for selecting the XML elements, and defining with OWL classes the elements are mapped to. The target RDF data is generated by processing these input mappings.

Vrandecic et al. (2005) suggests using a normalised form of RDF/XML by specifying a restricted form of DTDs that generate normalised XML format and again relying on standard XML processing tools for subsequent transformations. The provided DTD is used to generate SPARQL queries that access the RDF data and the system then relies on post-processing of the SPARQL query results to generate the desired output. The use of DTDs and automatic generation of SPARQL queries allows to leverage the existing XML users that are not familiar with RDF technologies.

[8] `http://rdfweb.org/people/damian/treehugger/index.html`, retrieved on 2012/07/17.
[9] `http://librdf.org/`, retrieved on 2012/07/17.
[10] `http://wasab.dk/morten/blog/archives/2004/05/30/transforming-rdfxml-with-xslt`, retrieved on 2012/07/17.
[11] `http://code.google.com/p/oort/wiki/Grit`, retrieved on 2012/07/17.

Also catering for SQL queries, Fischer et al. (2011) present a translation of both SQL and SPARQL queries into XQuery. Again the translation of SPARQL to XQuery operates on a normalised form of RDF/XML and thus a data translation step is required. A similar approach is taken for translating relational data into XML and then rewriting SQL to XQuery. In this paper the authors do not present an extended syntax language for the combination of data in the different formats and rather rely on the translation of data into XML.

RDF Twig (Walsh, 2003) suggests XSLT extension functions that provide views on the sub-trees of an RDF graph. The main idea of RDF Twig is that while RDF/XML is hard to navigate using XPath, a subtree of an RDF graph can be serialised in more useful forms that facilitate navigation. As such the authors provide XSLT extension functions that create different views of parts of the input RDF.

Several other approaches aim at automatically translating an XML Schema into an equivalent OWL ontology (Bohring and Auer, 2005; Rodrigues et al., 2008), focusing on mapping XML elements to OWL classes and properties. However in XSPARQL, we are focusing on translation and integration of instance data, rather than aiming to provide a semantic interpretation for XML data.

While not catering for the integration of XML, several other approaches focus on mapping relational data to RDF. For instance, D2R Server (Bizer, 2003) and D2R Map or Triplify (Auer et al., 2009) enable the conversion between RDB data and RDF. Large commercial database companies are also providing solutions for RDF triple stores, such as Oracle (Das and Srinivasan, 2009) and Virtuoso (Erling and Mikhailov, 2007). Most of these projects assume a fixed translation schema where, for instance, database tables are translated into RDFS classes and table attributes are represented as properties.

Language Integration

In this category of proposals we include the systems that consider the integration and/or expansion of query languages that allow the querying different formats without requiring the translation of data from the original formats.

Berrueta et al. (2008) presents a framework that facilitates SPARQL queries to be performed from XSLT: XSLT+SPARQL. It adds functions to XSLT that provide the ability to query SPARQL endpoints and uses standard XSLT to process the SPARQL XML results format. Similar to our current implementation, this relies on a clear separation between the SPARQL query and XSLT parts of the query.

Some proposals suggest compiling a SPARQL query to XSLT or XQuery. Bikakis et al. (2009) translate each SPARQL query into an XQuery using a mapping from OWL to XML Schema. The translation from SPARQL to XQuery is guided by the provided mapping (which can be automatically generated by a separate system) and thus allows the use of the SPARQL query language to access legacy XML data without the need to perform data translation.

Similarly Groppe et al. (2008) proposes to embed SPARQL into XSLT or XQuery, by presenting extensions to these languages that enable to query RDF data. In this proposal each SPARQL query is also translated into an equivalent XQuery. This language is very close to the XSPARQL language but however it requires converting the RDF data to XML according to a predefined schema. Assuming the queried dataset is available beforehand, this translation introduces an overhead to the query and, in case the dataset is not available for example due to being stored behind a SPARQL endpoint, such translation is not possible. In Chapter 5, we present some benchmark comparisons between an implementation of this language (provided to us by the authors) and our implementation of the XSPARQL language.

Ding and Buxton presented another approach to translate SPARQL into XQuery at the 2011 Semantic Technology Conference.[12] This rewriting generates XQuery specifically tailored for the Marklogic Server

[12] `http://semtech2011.semanticweb.com/sessionPop.cfm?confid=62&proposalid=4015`, retrieved on 2012/07/17.

XML database engine,[13] which incorporates RDF triples by using an internal XML representation.

Part of the CORESE Semantic Web framework,[14] Corby et al. (2009) provides extensions of SPARQL to process SQL, XPath, and XSLT in SPARQL queries. The authors also define an XSLT extension function that allows to evaluate SPARQL queries and integrate the query result into the XSLT processing. The implementation of these extensions is based on the CORESE framework, which employs caching mechanisms for the input XML and RDF documents. This approach is again similar to XSPARQL however the choice here was to extend SPARQL and XSLT, opposed to XSPARQL's extension of XQuery.

The Saxon XQuery engine (which we are using in our implementation) provides extension functions that allow to execute SQL queries and represent the results of the query as XML, easily incorporating them into the XQuery or XSLT query. This feature follows a similar implementation as XSPARQL but however does not provide the extend syntax as XSPARQL. The extension function executes a SQL query, although the functionality of injecting variable values provided by XSPARQL can be done, this task is left in charge of the query writer.

The nSPARQL query language (Pérez et al., 2008) proposes to extend SPARQL with navigational capabilities using nested regular expressions. With this addition, the language is sufficiently expressive to capture the semantics of RDFS. In addition to this, it introduces a number of graph navigation operators and adds the ability to selectively traverse the graph. This work is different than our current proposed approach for XSPARQL, but one of the possibilities for extending XSPARQL is to enable it to perform XQuery enriched SPARQL queries.

4.7. Conclusion

This chapter described the novel query language that we defined to tackle the integration of heterogeneous sources. We presented the syntax and semantics of the language, which are based on the syntax and semantics of the XQuery language. XSPARQL relies on the semantics of the other languages, SQL and SPARQL for querying the relational and RDF data and we also presented equivalences between the execution of queries in the different languages.

This query language forms the basis for a possible solution for the presented data integration scenario. In the next chapter we present our implementation of XSPARQL and tackle the problem of defining optimisations for the XSPARQL language, in an attempt to lower the query evaluation times for more complex queries, while the issue of representing meta-information in RDF is addressed in Chapter 6.

[13] `http://www.marklogic.com/products-and-services/marklogic-5/`, retrieved on 2012/07/17.
[14] `http://wimmics.inria.fr/corese`, retrieved on 2012/07/17.

5. XSPARQL Evaluation and Optimisations

This chapter describes our prototype implementation of the XSPARQL language presented in the previous chapter. We then describe a benchmark suite that will be used to evaluate our XSPARQL implementation. This benchmark is based on a widely used XML benchmark suite (XMark), and extends it to cater for XML and RDF data. The experimental evaluation will show that, in our current implementation, nested queries with an inner *SparqlForClause* present the highest overhead when compared to their XQuery counterpart.

To tackle this issue, Section 5.3 details different possible approaches for evaluating nested queries in our prototype and compares these approaches regarding their evaluation times. In Section 5.4 we present an overview of work related to the optimisations presented in this chapter.

5.1. Implementation

In this section we present our prototype implementation of the XSPARQL language, which translates an XSPARQL query into an XQuery query with interleaved calls to a relational database and/or a SPARQL engine. The architecture of our implementation is shown in Figure 5.1 and consists of the following main components: (1) a query rewriter, which turns an XSPARQL query into an XQuery; and (2) an (enhanced) XQuery engine for evaluating the rewritten XQuery. This enhanced XQuery engine relies on a SQL relational database and on a SPARQL engine, for accessing the heterogeneous data sources from within the rewritten XQuery.

We implement the XSPARQL language syntax and query rewriter by using the ANTLR parser generator,[1] which produces an XQuery with calls to the SQL and SPARQL engines. For the XQuery engine we use Saxon[2] and use the ARQ SPARQL engine[3] for querying the RDF data. As for accessing relational databases, we rely on a JDBC interface to the relational database and we have tested the connection to MySQL,[4] PostgreSQL,[5] and Microsoft SQL Server.[6] This interface between the different engines is implemented using the Saxon Extension API, which allows to create custom XQuery functions associated with Java methods. The functions that the rewritten queries use to access the relational and RDF data are called `xsp:sqlCall` and `xsp:sparqlCall`, which translate a *SQLForClause* or a *SparqlForClause* into a SQL or SPARQL `select` query respectively, and evaluate it, returning the results according to the types presented in Section 4.2.1.[7] However, instead of implementing all the newly introduced types as custom types in XQuery, we reuse the XML Schema of the SPARQL Query Results XML Format,[8] where the `sr:binding` type corresponds directly to XSPARQL's `RDFTerm` type. An `RDFGraph`, e.g. the result

[1] `http://www.antlr.org/`, retrieved on 2012/07/17.
[2] `http://saxon.sourceforge.net/`, retrieved on 2012/07/17.
[3] `http://jena.sourceforge.net/ARQ/`, retrieved on 2012/07/17.
[4] `http://www.mysql.com/`, retrieved on 2012/07/17.
[5] `http://www.postgresql.org/`, retrieved on 2012/07/17.
[6] `http://www.microsoft.com/sqlserver/`, retrieved on 2012/07/17.
[7] In the produced XQuery expressions we assume the reserved namespace prefix `xsp:` associated with `http://xsparql.deri.org/demo/xquery/xsparql.xquery`. This prefix is not allowed in an XSPARQL query and is used not only as the namespace for the XQuery functions `xsp:sqlCall` and `xsp:sparqlCall` but also as the namespace for any auxiliary variables introduced by the rewriting, effectively avoiding clashes with variables from the XSPARQL query.
[8] See `http://www.w3.org/2007/SPARQL/result.xsd`, we assume this schema is associated with the namespace prefix `sr`.

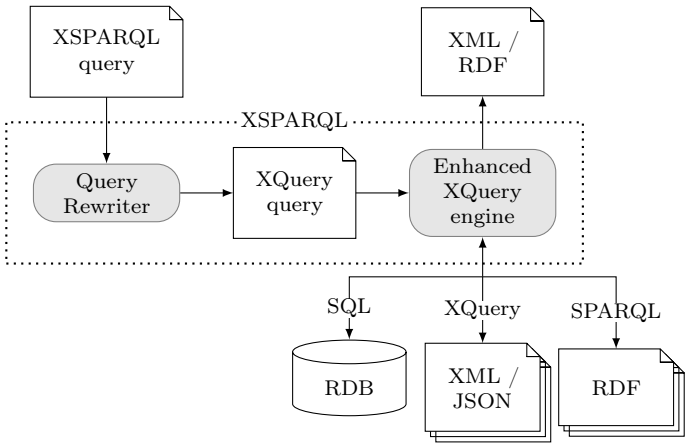


Figure 5.1.: XSPARQL implementation architecture

of a *ConstructClause*, is serialised using Turtle syntax by building the output as `xs:string`. The remaining types `RDFDataset` and `RDFNamedGraph` are adapted accordingly.

A more general form of using a SPARQL engine would be to rely on a SPARQL endpoint, as presented in the initial XSPARQL prototype described by Akhtar et al. (2008). However, by using Saxon's extension mechanism the query engines are more tightly integrated and allow for a more efficient communication of results (opposed to using a SPARQL endpoint via HTTP). As we will describe later in Section 5.1.1, we can still rely on this feature if necessary, and we use it for the implementation of the remote endpoint feature that allows us to mimic SPARQL 1.1 `SERVICE` feature.

Blank Node Matching in Nested Queries The `xsp:sparqlCall` function also implements the *matching blank nodes in nested queries* feature (as described in Section 4.2.2), for which we rely on custom Java code that uses the ARQ API to preserve blank node labels in consecutive SPARQL calls over the same dataset. The custom Java code maintains a *stack* of the previously used datasets during the query execution: upon the execution of a *SparqlForClause* with a *DatasetClause*, the code stores the blank node identifiers in the dataset and when executing a *SparqlForClause* without an explicit *DatasetClause*, we use the first element of the stack as the implicit dataset along with its existing blank node identifiers.

Creating Distinct Blank Nodes in *ConstructClause*s In the XSPARQL semantics (Section 4.2.4), we use the new globalPosition dynamic environment component to cater for creating fresh blank node identifiers for each instantiation of the *ConstructClause*. In our implementation we rely on *position variables* in XQuery `for` expressions (cf. Section 3.2.3) for generating the distinct identifiers. In the query rewriting step we normalise all the `for` expressions to include a position variable and also keep a list of all previous position variables in the query. Hence, when a blank node is found in a *ConstructClause*, we can generate the blank node label based on the label provided in the query and the values of the existing position variables.

Next we present how *SQLForClause*s, *SparqlForClause*s and *ConstructClause*s are processed by using what we call *rewriting functions*, which operate on syntactic objects of XSPARQL and return an XQuery expression.

5.1.1. *SQLForClause* and *SparqlForClause*

Our implementation defers the SQL and SPARQL query fragments to the respective external engines and extracts the bindings for the existing variables from the returned XML results document. For the

definitions of the rewriting functions, let $\mathbb{XS}$ and $\mathbb{XQ}$ denote the set of all XSPARQL and XQuery core expressions, respectively. The rewriting function $tr\colon \mathbb{XS} \to \mathbb{XQ}$ details our translation from XSPARQL to XQuery core. We now describe the rewriting function for the translation of *SparqlForClause*s, given an XSPARQL expression Q of form

```
for Vars DatasetClause
WhereClause SolutionModifier                    (Q1)
return ExprSingle
```

then $tr(Q)$ is defined as the XQuery Core expression

$tr(Q) =$

(1) `let $xsp:results := xsp:sparqlCall(` `select` *Vars DatasetClause* *WhereClause SolutionModifier* `) return`

(2) `for $xsp:result at $xsp:posvar in $xsp:results//sr:result return`

(3) `let` $\$v :=$ `$xsp:result/sr:binding[@name =` v`]/* return` for each $\$v \in$ *Vars*

(4) *ExprSingle*

That is, we implement the *fs:sparql* formal semantics function by translating Q into a SPARQL `select` query, which is then executed by the custom runtime function `xsp:sparqlCall`. that returns the result in SPARQL's XML result format. This is represented in line (1) of the rewritten query. The `for` expression in line (2) selects all solutions from the XML representation of the query results, while the `let` expressions in line (3) assign the result value of each variable to XSPARQL variables. Finally, the `return` expression *ExprSingle* of line (4) is evaluated with the new variables available.

For XSPARQL *SQLForClause*s of the form

```
for AttrSpec_1 as $Var_1, ..., AttrSpec_n as $Var_n
RelationList SQLWhereClause                     (Q2)
return ExprSingle
```

then $tr(Q)$ is defined as the XQuery Core expression

$tr(Q) =$

(1) `let $xsp:results := xsp:sqlCall(` `select` $AttrSpec_1, \ldots, AttrSpec_n$ *RelationList SQLWhereClause* `) return`

(2) `for $xsp:result at $xsp:posvar in $xsp:results//sr:result return`

(3) `let` $\$Var_i :=$ for each $AttrSpec_i, \$Vari \in AttrSpec_1$ `as` $\$Var_1, \ldots, AttrSpec_n$ `as` $\$Var_n$
`$xsp:result/sr:binding[@name =` $AttrSpec_i$`]/* return`

(4) *ExprSingle*

Furthermore, in case the XSPARQL specifies the attribute selection as a '`FOR *`' the translation function requires access to the input relational database during the rewriting in order to determine the relation attributes and the names of the XSPARQL variables to be generated.

Implementation of the XSPARQL Semantics

The presented `xsp:sparqlCall` function also implements XSPARQL's BGP matching, as described in Section 4.2, by replacing any previously assigned variables in the SPARQL query with their current value according to the rules presented in Definition 4.9. This behaviour implements XSPARQL's BGP matching while relying on an off-the-shelf SPARQL engine. In Section 5.1.3 we present the formal correspondence between the variable replacement approach and XSPARQL's semantics. This replacement of variables can be statically determined during the query rewriting step and generate the respective SQL or SPARQL

query string (the parameters to the `xsp:sqlCall` and `xsp:sparqlCall` functions, respectively) by using XQuery's `fn:concat` function. The `fn:concat` allows for an arbitrary number of arguments and when executed concatenates the string value resulting from the evaluation of each argument. When parsing a *SparqlForClause* we have access to the set of previously declared variables and, whenever we find a variable it is possible to determine whether to replace it by its previously assigned value or keep it as a variable. If the variable has been previously declared, the variable name is inserted as an argument of the `fn:concat` function, which upon evaluation accesses the value of the variable and use it in the creation of the `select` query. If the variable is fresh, i.e. has not been declared before, we leave the variable name as a (quoted) string within the `fn:concat`, which effectively postpones the evaluation of the variable to the SPARQL engine.

Example 5.1 (`select` query generation)**.** Consider the following simple XSPARQL query:

```
let $name := "Nightwish"
for * where { $band foaf:name $name }
return $band
```

The rewritten code that generates the SPARQL `select` query is as follows:

```
fn:concat("SELECT $band where { ", "$band", " foaf:name ", $name, "}")
```

Note that the '`for *`' has also been replaced to select only the *unbound* variables in the *WhereClause*.

For *SQLForClause*s we follow a similar approach but since SQL does not allow for $-prefixed variable names, we always leave the variable name unquoted, which means that for *SQLForClause*s all variables used in *SQLWhereClauses* must be previously bound.

Querying External SPARQL Endpoints Since our implementation of XSPARQL rewrites *SparqlForClause*s into SPARQL queries, we can execute the rewritten SPARQL query in different ways: the typical way is to use a local instance of the ARQ engine to execute the query. One of the new features of SPARQL 1.1 (presented in Section 3.3) is the `SERVICE` keyword, which specifies that the following subquery will be executed in a remote SPARQL endpoint. In XSPARQL we can also enable this behaviour by specifying the '`endpoint`' URI in a *SparqlForClause*, after the *DatasetClause*. This instructs the XSPARQL engine to use the remote endpoint specified by URI for executing the query and incorporate the bindings into the query as usual. As opposed to SPARQL 1.1, which does not allow to inject bindings of results into the `SERVICE` subquery,[9] our variable replacement operation allows to inject values from the outer query into the inner query. This feature allows to write queries that are otherwise unavailable or impractical in SPARQL, either by design restrictions of the language or practical restrictions of SPARQL endpoints (as illustrated in the following example).

Example 5.2 (Querying Remote SPARQL Endpoints)**.** Consider as an example we want to retrieve from DBPedia persons that have the same birthday as Marco Hietala. For this we first need to retrieve Marco's birthday (we are taking this information from DBPedia but we could rather use the artists personal FOAF file) and then retrieve from DBPedia the persons with the same birthday. Queries 5.1 and 5.2 present possible XSPARQL and SPARQL versions of this query, respectively. These queries both involve querying the remote DBPedia SPARQL endpoint. The SPARQL version (Query 5.2) quickly runs into limits imposed by the DBPedia SPARQL endpoint, since the `SERVICE` nested query attempts to retrieve the all persons that have any birthday specified, due to not being

[9] Please note that the `BINDINGS` can only take fixed values for variables, preventing to use results from the execution of other parts of the query

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix : <http://example.org/>
prefix dbpedia-owl: <http://dbpedia.org/ontology/>

let $MB := for * from <http://dbpedia.org/resource/Marco_Hietala>
           where { [ dbpedia-owl:birthDate $B ]. }
           return $B

for * from <http://dbpedia.org/>
endpoint <http://dbpedia.org/sparql>
where { [ dbpedia-owl:birthDate $B; foaf:name $N ] . filter ( regex(str($B),str($MB))
    ) }
construct { <http://dbpedia.org/resource/Marco_Hietala> :sameBirthDayAs $N }
```

Query 5.1: Querying a remote endpoint with XSPARQL

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT $M $MB $B
FROM <http://dbpedia.org/resource/Marco_Hietala>
WHERE { [ dbpedia-owl:birthDate $MB ]

  SERVICE <http://dbpedia.org/sparql> {
    SELECT $B $N WHERE {
       [ dbpedia-owl:birthDate $B ; foaf:name $N ]. } }

  FILTER ( regex(str($B), str($MB)) )
}
```

Query 5.2: Querying a remote endpoint with SPARQL

aware of the bindings of variable $\$MB$. On the other hand, in the XSPARQL version in Query 5.1 the inner `endpoint` query only retrieves the required birthdays.

5.1.2. *ConstructClause*

As for the construction of RDF graphs (i.e. whenever the *ReturnClause* is a *ConstructClause*), our implementation XQuery rewriting produces a string in Turtle syntax, where we ensure that each generated RDF triple is syntactically valid. For this we rely on a number of additional auxiliary XQuery functions: firstly, the function `xsp:rdfTerm`($\$VarName$) (presented in Figure 5.2a), when given a variable of type `RDFTerm`, returns the correctly formatted RDF term (according to the Turtle syntax) of $\$VarName$. Next, the `xsp:validTriple` presented in Figure 5.2b implements the semantics function *fs:validTriple* by calling the `xsp:rdfTerm` function to correctly format triples to Turtle syntax. This function further uses the auxiliary functions `xsp:validSubject`, `xsp:validPredicate` and `xsp:validObject` that determine, according to the RDF semantics, if their argument is a valid subject, predicate, or object, respectively.

Our implementation of the *fs:skolemConstant* function, that ensures blank nodes in `construct` expressions are distinct between different solutions, consists of appending the position variables from all the surrounding `for` expressions to the respective blank node identifier using "_" as a separator. This is represented by the following rewriting function

$$tr_{sk}(\$BNodeName, \{\ \$PosVar_1, \cdots, \$PosVar_n\ \}) = \\ \texttt{fn:concat}(\texttt{"\_:"}, \$BNodeName, \texttt{"\_"}, \$PosVar_1, \cdots, \texttt{"\_"}, \$PosVar_n)\ \ .$$

```
declare function xsp:rdfTerm($VarName) {
  typeswitch $VarName
  case $e as literal
    let $DT := data($e/@datatype)
    let $L:= data($e/@xml:lang)
    return concat("""", $e,
          if($L) then concat("@", $L) else "",
          if($DT) then concat("^^<", $DT,">")
              else "",
        """")
  case $e as bnode return concat("_:", $e)
  case $e as uri return concat("<", $e, ">")
  default return "" };
```

(a) `xsp:rdfTerm` function

```
declare function xsp:validTriple($sub,
      $pred, $obj) {
  if(xsp:validSubject($sub)
     and xsp:validPredicate($pred)
     and xsp:validObject($obj))
  then concat(xsp:rdfTerm($sub), " ",
              xsp:rdfTerm($pred), " ",
              xsp:rdfTerm($obj), ".")
  else "" };
```

(b) `xsp:validTriple` function

Figure 5.2.: Implementation functions example

Finally, the function `xsp:evalCT` implements *fs:evalCT* by simply concatenating all the triples generated by the `xsp:validTriple` function to a string representation of the RDF graph to be constructed.

Implementation of Constructed Datasets As described in Section 4.1, it is possible to assign the result of a `construct` query to an XSPARQL variable, which can later be used in the *DatasetClause* of a *SparqlForClause*. In order to make this constructed graph available to the ARQ SPARQL engine, we need to materialise it as a temporary file and specify this temporary file's location in the SPARQL query. To enable this feature, during the query rewriting step, whenever we find a *ConstructClause* assigned to an XSPARQL variable, we create a temporary RDF file with the result of the `construct` expression represented as Turtle and assign the local path of this generated file to the XSPARQL variable.

5.1.3. Soundness & Completeness of the Implementation

We next present the equivalence between our implementation of the XSPARQL language and the XSPARQL semantics presented in Section 4.2. We start by presenting a lemma stating that the results of the evaluation of a BGP P under XSPARQL BGP matching semantics can be determined based on the results of evaluating $\mu(P)$ (cf. Definition 3.7) under SPARQL semantics. Similar correspondence for *SQLForClause*s is presented was Lemma 4.2.

Lemma 5.1. *Let P be a BGP, D a dataset and μ the XSPARQL instance mapping of P. Considering $P' = \mu(P)$, we have that $eval_{xs}(D, P, \mu) = eval(D, P') \bowtie \{ \mu \}$.*

Proof: Since, according to the variable substitution operation we have that $vars(P') = vars(P) \setminus dom(\mu)$, we also have that $vars(P') \cap dom(\mu) = \emptyset$ and it follows directly from Lemma 4.3 that $eval_{xs}(D, P, \mu) = eval(D, P') \bowtie \{ \mu \}$. □

The following result presents the equivalence of our implementation function *tr* and the XSPARQL semantics.[10]

Proposition 5.2. *Let Q be a SparqlForClause of form* (Q1) *or a SQLForClause of form* (Q2) *and dynEnv the dynamic environment of Q, then $dynEnv \vdash Q \Rightarrow Val$ if and only if $dynEnv \vdash tr(Q) \Rightarrow Val$.*

Proof: We present here only the proof for *SparqlForClause*s of form (Q1), the proof for *SQLForClause*s is analogous.

[10]Please note that, for presentation purposes, we are omitting the initial *empty line* in case the proof trees require no premises and the variable expansion premises.

($\Leftarrow$) Let us show that if $\text{dynEnv} \vdash tr(Q) \Rightarrow Val$ then $\text{dynEnv} \vdash Q \Rightarrow Val$. The evaluation of Q consists of the application of Rule (D7) as

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_j)}{\text{dynEnv} \vdash fs{:}dataset(DatasetClause) \Rightarrow Dataset}}{\text{dynEnv} \vdash fs{:}sparql\left(\begin{array}{l} Dataset, WhereClause, \\ SolutionModifier \end{array}\right) \Rightarrow \mu_i^{xs}} \quad \cdots}{\text{dynEnv}_1^{xs} \vdash ExprSingle \Rightarrow Value_i}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } \$Var_1 \cdots \$Var_n \ DatasetClause \\ WhereClause \ SolutionModifier \\ \texttt{return } ExprSingle \end{array} \Rightarrow Value_1 \cdots Value_m}$$

where, for each μ_i^{xs},

$$\text{dynEnv}_1^{xs} = \text{dynEnv} \ + \text{activeDataset}(Dataset) \ + \text{globalPosition}((Pos_1, \cdots, Pos_j, i)) \ + \text{varValue}\left(\begin{array}{l} Var_1 \Rightarrow fs{:}value(\mu_i^{xs}, Var_1)\,; \\ \cdots\,; \\ Var_n \Rightarrow fs{:}value(\mu_i^{xs}, Var_n) \end{array}\right) . \tag{T1}$$

Let μ_C be the XSPARQL instance mapping of the expression context that includes dynEnv and Ω_{tr} the pattern solution resulting from the evaluation of the `xsp:sparqlCall` function, i.e. $\Omega_{tr} = eval(DatasetClause, P)$, where P is the rewriting of *WhereClause* according to μ_C. Furthermore, let $\mu_i \in \Omega_{tr}$ be the solution mapping from which *Val* is generated, i.e. there exists some dynamic environment dynEnv^{tr} based on dynEnv and extended with the variable bindings from μ_i such that $\text{dynEnv}^{tr} \vdash ExprSingle \Rightarrow Val$.

Consider $\Omega_{xs} = eval_{xs}(DatasetClause, WhereClause, \mu_C)$ as the solution sequence resulting from the evaluation of the *fs:sparql* function. As we know from Lemma 5.1, $\Omega_{xs} = \Omega_{tr} \bowtie \{\, \mu_C \,\}$ and thus there must exist a solution mapping $\mu_{xs} \in \Omega_{xs}$ such that $\mu_{xs} = \mu_i \bowtie \mu_C$. From (T1) we know that there exists a dynamic environment dynEnv^{xs} that results from extending dynEnv with the variable bindings from μ_{xs} and thus this environment will also contain all the variable mappings from μ_i (and from dynEnv^{tr}, respectively). Since we know that $\text{dynEnv}^{tr} \vdash ExprSingle \Rightarrow Val$, we also have that $\text{dynEnv}^{xs} \vdash ExprSingle \Rightarrow Val$ and thus $\text{dynEnv} \vdash Q \Rightarrow Val$.

($\Rightarrow$) Next we will show that if $\text{dynEnv} \vdash Q \Rightarrow Val$ then $\text{dynEnv} \vdash tr(Q) \Rightarrow Val$. We present the proof tree for each of the XQuery core expressions in the $tr(Q)$ rewriting. The proof trees are presented for each line of the $tr(Q)$ rewriting and, in each proof tree, *Expr* corresponds to the XQuery expressions of the following lines.

`let` expression of line (1):

$$\dfrac{\dfrac{\text{dynEnv} \vdash \texttt{xsp:sparqlCall}\left(\begin{array}{l} \texttt{select } Vars \ DatasetClause \\ WhereClause \ SolutionModifier \end{array}\right) \Rightarrow \Omega_{tr}}{\text{dynEnv}_1^{tr} \vdash Expr \Rightarrow Res}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{let } \$\texttt{xsp:results} := \\ \quad \texttt{xsp:sparqlCall}\left(\begin{array}{l} \texttt{select } Vars \ DatasetClause \\ WhereClause \ SolutionModifier \end{array}\right) \\ \texttt{return } Expr \end{array} \Rightarrow Res}$$

where

$$\text{dynEnv}_1^{tr} = \text{dynEnv} \ + \text{varValue}(\texttt{xsp:results} \Rightarrow \Omega_{tr}) \quad . \tag{T2}$$

`for` **expression of line (2):**

$$\dfrac{\dfrac{\text{dynEnv}_1^{tr} \vdash \$\texttt{xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}_2^{tr} \vdash \mathit{Expr} \Rightarrow \mathit{Res}_i} \quad \cdots}{\text{dynEnv}_1^{tr} \vdash \begin{array}{l}\texttt{for } \$\texttt{xsp:result at } \$\texttt{xsp:posvar} \\ \quad \texttt{in } \$\texttt{xsp:results//sr:result} \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}_1, \cdots, \mathit{Res}_n}$$

where $\text{dynEnv}_2^{tr} = \text{dynEnv}_1^{tr} + \text{varValue}\left(\begin{array}{l}\texttt{xsp:result} \Rightarrow \mu_i; \\ \texttt{xsp:posvar} \Rightarrow i\end{array}\right)$

`let` **expressions of lines (3)–(4).** Here we consider all the `let` expressions represented by line (3), where $\$v \in \mathit{Vars}$, i.e. this rule is repeated for each $\$v \in \mathit{Vars}$:

$$\dfrac{\dfrac{\text{dynEnv}_2^{tr} \vdash \$\texttt{xsp:result/sr:binding}[\texttt{@name} = v]/* \Rightarrow V}{\text{dynEnv}_3^{tr} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Res}}}{\text{dynEnv}_2^{tr} \vdash \begin{array}{l}\texttt{let } \$v := \$\texttt{xsp:result/sr:binding}[\texttt{@name} = v]/* \\ \texttt{return } \mathit{ExprSingle}\end{array} \Rightarrow \mathit{Res}}$$

where $\text{dynEnv}_3^{tr} = \text{dynEnv}_2^{tr} + \text{varValue}(v \Rightarrow V)$

Consider the dynamic environment dynEnv_i^{xs} such that $\text{dynEnv}_i^{xs} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Val}$ where, as we know from (T1), dynEnv_i^{xs} extends dynEnv with the bindings from a solution mapping μ_i^{xs}. Furthermore, consider μ_C, Ω_{xs}, and Ω_{tr} as per the ($\Leftarrow$) part of the proof.

From the proof trees of $tr(Q)$ we can see that the `let` expression from line (1) extends dynEnv with the value for the reserved variable $\$\texttt{xsp:results}$, which cannot be included in Q. The `for` expression from line (2) iterates over all the solution mappings $\mu_{tr} \in \Omega_{tr}$ and, as we know from Lemma 5.1, $\Omega_{xs} = \Omega_{tr} \bowtie \{\mu_C\}$. Since μ_C is created based on dynEnv.varValue, all the variable bindings from μ_C are already included in dynEnv and all solution mappings $\mu_{tr} \in \Omega_{tr}$ are guaranteed to be compatible with μ_C and thus we have that $\mu_i^{xs} \in \Omega_{tr}$.

Finally, the `let` expressions from lines (3) and (4) ensure that there exists a dynEnv_2^{tr} such that dynEnv_2^{tr}.varValue contains all the variable bindings from μ_i^{xs}, and we have that $\text{dynEnv}_2^{tr} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Val}$ and $\text{dynEnv} \vdash tr(Q) \Rightarrow \mathit{Val}$. □

5.2. The XMarkRDF benchmark

For the evaluation of our implementation we created a benchmark suite based on the XMark benchmark suite (Schmidt et al., 2002), which according to Afanasiev and Marx (2008) is the most widely used benchmark suite for XQuery. It provides a data generator that produces XML data simulating an auction website (including information about persons and items they bid for) and includes 20 XQuery queries, henceforth referred to as q_1 to q_{20}, over this generated data.

In order to benchmark the XSPARQL language we also require data in the relational and RDF formats, hence we provide transformations (in fact, using XSPARQL queries) from the XMark XML datasets into RDF triples and a relational instance, following a manually created schema for representing the XMark data. These transformations replicate all the data in the original XMark datasets as RDF triples and relational tuples. Next, we converted the XMark queries into corresponding XSPARQL queries using *SparqlForClause*s and *SQLForClause*s to access the RDF data and the relational database, respectively. We call this new benchmark suite the XMarkRDF benchmark.

We have made two changes to the original XMark queries: (1) SPARQL queries do not guarantee any default ordering, hence all original XMark queries were declared unordered – as a consequence the XQuery engine is not required to follow document order when executing the query; and (2) we added the

Table 5.1.: XMark (and variants) benchmark dataset description

Scaling factor	Persons	Categories	XMark Size (MB)	XMarkRDF Size (MB)	XMarkRDF # Triples	XMarkRDB Size (MB)	XMarkRDB # Tuples
0.01	255	10	1.1	1.2	14745	1	4112
0.02	510	20	2.3	2.3	27519	2	7799
0.05	1275	50	5.8	5.8	70859	5	20190
0.10	2550	100	11.7	12.4	142721	10	40183
0.20	5100	200	23.5	24.9	283639	20	80622
0.50	12750	500	58.0	61.7	706723	50	200496
1.00	25500	1000	116.5	124.8	1414469	101	400620

Table 5.2.: XMarkRDF$_{S2XQ}$ dataset and translation times

Scaling factor	Dataset size (MB)	Translation times (sec)
0.01	3.3	18.94
0.02	6.4	18.30
0.05	16.1	26.08
0.10	32.7	39.01
0.20	65.3	62.35
0.50	162.3	143.35
1.00	326.2	329.93

external variables $\$xml$ and $\$rdf$ in the XQuery and XSPARQL queries as parameters used to specify the URI identifying the input benchmark instance.

We also included in our own comparisons the SPARQL2XQuery system (Groppe et al., 2008), which is similar in spirit to XSPARQL. While the SPARQL2XQuery language allows to perform similar queries to the RDF and XML fragment of the XSPARQL language, the implementation follows a different approach to integrate the XML and RDF data: rather than performing interleaved calls to a SPARQL engine, the SPARQL2XQuery system relies on translating the RDF data into a pre-defined XML format and transforming SPARQL queries into equivalent XQuery over this pre-defined XML format. The translated queries can be directly executed using a native XQuery engine. We focussed our experimental evaluation on query response time rather than on data transformation time, and as SPARQL2XQuery requires an additional translation step from RDF to a custom RDF/XML format, we converted the XMarkRDF RDF data into the format required by the SPARQL2XQuery system. We denote these new datasets, containing the RDF/XML format required for the SPARQL2XQuery, by XMarkRDF$_{S2XQ}$.

Using the data generators and translators, provided by the XMark benchmark and the XSPARQL translation to RDF (as presented in Section 5.3), we created datasets with scaling factors of 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, and 1.0 and translated them into XMarkRDF and XMarkSQL. An overview of the generated data is presented in Table 5.1, including the number of persons and item categories modelled, dataset sizes,[11], the number of relational tuples and RDF triples.

Furthermore, we converted the XMarkRDF datasets into the RDF/XML format required by the SPARQL2XQuery system. The resulting dataset sizes and translation times for the different scaling factors of the XMarkRDF dataset are presented in Table 5.2.

[11]For the dataset sizes we determined the dataset size based on a Turtle representation of the RDF graph and the SQL INSERT statements that populate the database.

Table 5.3.: Query response times (in seconds) of the 2MB dataset. Query rewriting error (*err*).

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
XQ	0.71	**0.70**	**0.77**	0.74	0.71	0.70	**0.72**	**1.11**	**1.12**	**0.99**
XS^{rdb}	**0.19**	0.75	1.50	**0.22**	**0.26**	**0.38**	0.85	1.27	1.56	1.62
XS^{rdf}	3.06	3.29	3.43	3.08	3.18	3.32	3.94	293.62	292.84	16.92
S2XQ	1.06	17.41	*err*	65.13	1.00	0.98	*err*	1.28	11.57	309.21

	q_{11}	q_{12}	q_{13}	q_{14}	q_{15}	q_{16}	q_{17}	q_{18}	q_{19}	q_{20}
XQ	**0.97**	**0.94**	0.73	0.73	**0.69**	**0.72**	0.72	0.74	0.78	**0.74**
XS^{rdb}	2.33	1.81	**0.28**	**0.56**	*err*	*err*	**0.59**	**0.36**	**0.69**	1.20
XS^{rdf}	295.19	55.55	3.20	3.15	*err*	*err*	3.24	3.24	3.92	6.75
S2XQ	102.42	70.89	7.84	1.01	1.42	7.77	8.54	6.10	13.43	—

5.2.1. Experimental Setup

The benchmark system consists of a dual core Intel XEON E5606 2.13GHz, 64GB memory running a 64 bit installation of Debian 6.0.3 (stable distribution). For the XQuery engine, we rely on Saxon version 9.4 Home Edition and Java version 1.6.0 64 bit. For evaluating SPARQL queries we used ARQ 2.8.7. We ran each query with a timeout of 10 minutes per query and with the Java Heap size set to 1GB. Each query was run 10 times and the response time was measured using GNU time 1.7. For each query we discard the fastest and slowest response time and calculate the average of the remaining times. From this result we deduce the process startup time, determined by following the same procedure and executing an empty query.

For the evaluation we defined the following run configurations:

XQ: original XQuery queries, evaluated using the Saxon engine;
XS^{rdf}: using the XSPARQL implementation over the XMarkRDF datasets (translated data and queries);
XS^{rdb}: using the XSPARQL implementation over the XMarkRDB datasets stored on a PostgreSQL 8.4.11 relational database management system (translated data and queries); and
S2XQ: using the SPARQL2XQuery implementation over the translation of the XMarkRDF datasets into the required XML format (XMarkRDF_{S2XQ}).

5.2.2. Base System Results

In this section we present an experimental evaluation of our prototype presented in Section 5.1 using the novel XMarkRDF and XMarkRDB benchmark suites. We also compare our XSPARQL prototype with the SPARQL2XQuery engine, an implementation of the direct translation of SPARQL to XQuery presented by Groppe et al. (2008).

The response times of the *XQ*, XS^{rdb}, XS^{rdf}, and *S2XQ* runs for the benchmark queries over the 2MB dataset size are shown in Table 5.3.[12] We present the 2MB dataset as it is the largest dataset our XS^{rdf} implementation can process within the time limit of 10 minutes. Both the data and query translation times for the *S2XQ* configuration are not included in the presented results since this process can be done a priori. The *XQ* response times are presented as a baseline measure, however it is noteworthy that these queries do not cater for our heterogeneous data sources scenario. The XS^{rdb} configuration often

[12]Queries q_{15} and q_{16} involve applying an XPath expression to data that is stored in RDF or in the relational database as a string. Since parsing this string representation back into an XML element is not available to the Saxon HE engine we are using for benchmarking, these queries were considered as errors (*err*) for the XS^{rdf} and XS^{rdb} configurations. The errors in *S2XQ* were due to translation errors from the application that was provided to us.

Table 5.4.: Query response times (in seconds) of the 100MB dataset. Query rewriting error (*err*).

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
XQ	3.19	**3.27**	**3.39**	3.32	3.16	3.06	**3.07**	154.59	168.78	22.64
XS^{rdb}	**0.19**	120.92	45.78	**0.24**	**2.07**	**2.44**	3.64	**62.43**	**68.86**	**20.31**
XS^{rdf}	37.84	41.53	42.41	37.93	40.28	40.43	41.96	—	—	—
S2XQ	—	—	*err*	—	—	—	*err*	—	—	—

	q_{11}	q_{12}	q_{13}	q_{14}	q_{15}	q_{16}	q_{17}	q_{18}	q_{19}	q_{20}
XQ	**93.40**	**37.50**	3.34	**3.31**	**3.05**	**3.05**	3.35	3.59	**3.94**	**3.34**
XS^{rdb}	—	443.29	**1.68**	3.41	*err*	*err*	**2.90**	**2.52**	3.99	71.41
XS^{rdf}	—	—	40.07	39.30	*err*	*err*	40.90	41.55	41.58	—
S2XQ	—	—	—	—	—	—	—	—	—	—

presents the best response times undoubtably due to the underlying relational database system. Since the inner queries in nested queries access the primary keys of relations, their response time is fast. Table 5.4 presents the results for our largest dataset, where we can see that the XS^{rdb} approach is able to evaluate most of the queries (except q_{11}) within the time limit.

Table 5.3 shows that for most of the queries in the *S2XQ* runs are faster than the interleaved calls to a SPARQL engine in the XS^{rdf} runs. Even considering that the response times do not include the data translation times (presented in Table 5.2), this suggests that an implementation of XSPARQL where the SPARQL queries are translated into native XQuery is a viable alternative to interleaving calls to a SPARQL engine. However, for such translations to be possible we need access to the full RDF dataset to perform the query translation, which is not possible for example in the case where we are querying data behind a SPARQL endpoint. Another issue related to the implementation of the SPARQL2XQuery system is that response times deteriorate considerably for larger datasets. This was observed for all the queries in the benchmark and can be seen in the graphs of Figures 5.5 and 5.6.

Queries q_8–q_{12} have the highest execution times of all the benchmark queries (especially noticeable in the XS^{rdf} configuration) since they contain nested expressions. For these nested queries, our interleaved *SparqlForClause*s XSPARQL implementation can only handle small datasets: the 2MB dataset is the largest for which all queries finish within the time limit and for the 20MB dataset all queries result in a *timeout*.

Based on these results, we propose a set of different rewritings that aim at reducing the response times of nested queries.

5.3. Optimisations of Nested `for` Expressions

Following our current implementation of the XSPARQL language, this section presents different rewriting strategies for XSPARQL queries containing nested expressions. Based on the experimental evaluation results from the previous section, we are especially interested in nested expressions with an inner *SparqlForClause*, as the number of interleaved calls to the SPARQL engine can be reduced drastically by using these rewritings. Intuitively, these rewritings rely on executing the inner SPARQL query only once in an unbounded manner, and then either performing the a nested loop over the results of the queries directly in XQuery, or, if possible, transforming the nested queries into a single SPARQL query.

We start by presenting the definitions and conditions under we can perform these rewritings.

Definition 5.1 (Dependent Join)**.** *We call two nested XSPARQL* `for` *expressions (ForClause, SparqlFor-*

Clause, or SQLForClause), where the inner expression is a SparqlForClause and at least one variable in the inner expression is bound by the outer expression, a dependent join*. The shared variables between the* `for` *expressions are called* dependent variables*.*

Note that the strategies presented here are only applicable for dependent joins satisfying the following restrictions:

1. An explicit *DatasetClause* of the inner query needs to be statically determined i.e. it cannot be determined based on variables bound from the outer expression;
2. The return clause of the inner expression can not be a *ConstructClause*; and
3. The dependent variable in the inner query's graph pattern must be *strictly bounded* as defined next.

Definition 5.2 (Strict Boundedness)**.** *The set of* strictly bound variables *in a graph pattern P, denoted $bVars(P)$, is recursively defined as follows: if P is*

- *a BGP, then $bVars(P) = vars(P)$;*
- *$(P_1$ `and` $P_2)$, then $bVars(P) = bVars(P_1) \cup bVars(P_2)$;*
- *$(P_1$ `optional` $P_2)$, then $bVars(P) = bVars(P_1)$;*
- *$(P_1$ `union` $P_2)$, then $bVars(P) = bVars(P_1) \cap bVars(P_2)$;*
- *$($`graph` i $P_1)$, then $bVars(P) = bVars(P_1) \cup (\{ i \} \cap \mathbf{V})$; and*
- *$(P_1$ `filter` $R)$, then $bVars(P) = bVars(P_1)$.*

Informally, the dependent variables must occur (i) in a BGP, (ii) in every alternative of `union`s pattern, and (iii) it must also occur outside of the optional graph pattern in case of optionals. Strict boundedness essentially ensures that the join variable does not occur only in a `filter` expression, which would lead to problems in case the inner expression is called unconstrained, see below.

Next, we define the notion of inclusion of solution sequences.

Definition 5.3 (Solution sequence inclusion)**.** *Let Ω_1 and Ω_2 be solution sequences. We say Ω_1* is included in *Ω_2, denoted $\Omega_1 \preceq \Omega_2$, if for all solution mappings $\mu_1 \in ToMultiset(\Omega_1)$ there exists a solution mapping $\mu_2 \in ToMultiset(\Omega_2)$ such that $\mu_1 \subseteq \mu_2$.*

Please note that this definition extends the notion of subset between multisets by considering also the subset relation between their elements, i.e. solution mappings.

The following rewritings for the implementation of dependent joins can be grouped into two categories, depending whether the join is performed in XQuery or SPARQL. For performing the join in XQuery, we use already known join algorithms from relational databases, namely nested-loop joins. For performing the join in SPARQL, if the outer expression is a *SparqlForClause* we can implement the join by rewriting both the inner and the outer expressions into a single SPARQL call. In case the outer query consists of an XQuery *ForClause*, we can still consider this approach, but we need to convert the result of the outer XQuery *ForClause* to an RDF graph, for instance relying on a SPARQL engine that supports SPARQL Update (Gearon et al., 2012) to add this temporary graph to a triple store.

5.3.1. Dependent Join implementation in XQuery

The intuitive idea with these rewritings is, instead of using the naïve rewriting that performs one SPARQL query for each iteration of the outer expression, to execute only one unconstrained SPARQL query, before the outer query. The resulting sequence of SPARQL solution mappings is then joined in XQuery with the results of the outer expression, using one of the following strategies.

The straightforward way to implement the join over dependent variables directly in XQuery is by nesting two XQuery `for` expressions, much like a regular nested-loop join (Abiteboul, Hull et al., 1995)

in standard relational databases. The join consists of restricting the values of variables from the inner expression to the values taken from the current iteration of the outer expression.

Similar to Section 5.1, we will describe the implementation of this nested-loop join by means of the rewriting function opt_{nl}. We use $A \triangle B = (A \cup B) \setminus (A \cap B)$ to denote the symmetric difference of two sets A and B.

Let Q be an XSPARQL expression of form

$$
\begin{array}{ll}
(1) & \texttt{for } \$Var^{out} \texttt{ at } \$PosVar^{out} \texttt{ in } ExprSingle_1 \texttt{ return} \\
(2) & \quad \texttt{for } Vars^{in}\ DatasetClause\ WhereClause\ SolutionModifier \\
(3) & \quad \texttt{return } ExprSingle_2
\end{array}
\tag{Q3}
$$

the application of the rewriting function $opt_{nl}(Q)$ can be split into two cases:

- if $ExprSingle_1$ and $ExprSingle_2$ do not contain any occurrences of (Q3) then, assuming $Vars^{sp} = Vars(WhereClause)$, we have that:

$$
\begin{array}{ll}
opt_{nl}(Q) = & \\
(1) & \texttt{let } \$\texttt{xsp:results} := \texttt{xsp:sparqlCall}\left(\begin{array}{l} \texttt{select } \{\$Var^{out}\} \cup Vars^{in}\ DatasetClause \\ WhereClause\ SolutionModifier \end{array} \right) \texttt{return} \\
(2) & \texttt{for } \$Var^{out} \texttt{ at } \$PosVar^{out} \texttt{ in } ExprSingle_1 \texttt{ return} \\
(3) & \quad \texttt{for } \$\texttt{xsp:result at } \$\texttt{xsp:posvar\_in in } \$\texttt{xsp:results//sr:result return} \\
(4) & \quad \texttt{if } \left(join_{nl}\left(\begin{array}{l} \{\$Var^{out}\} \cap Vars^{sp}, \\ \$\texttt{xsp:result} \end{array} \right) \right) \texttt{ then} \\
(5) & \qquad \texttt{let } \$v := \$\texttt{xsp:result/sr:binding}[\texttt{@name} = v]/* \qquad \text{for each } \$v \in \{\ Var^{out}\ \} \triangle Vars^{sp} \\
(6) & \qquad ExprSingle_2 \\
(7) & \quad \texttt{else } ()
\end{array}
$$

- otherwise:

$$
\begin{array}{l}
opt_{nl}(Q) = \\
\quad opt_{nl}\left(\begin{array}{l} \texttt{for } \$Var^{out} \texttt{ at } \$PosVar^{out} \texttt{ in } opt_{nl}(ExprSingle_1) \texttt{ return} \\ \quad \texttt{for } Vars^{in}\ DatasetClause\ WhereClause\ SolutionModifier \\ \quad \texttt{return } opt_{nl}(ExprSingle_2) \end{array} \right)
\end{array}
$$

The auxiliary function $join_{nl}$ consists of an XPath expression that determines if an XQuery tuple stream is compatible with a SPARQL solution mapping. More specifically, this function considers two variables as compatible if their values are equal, the outer value is a blank node, or the inner value ($\$VarRes_i$) is unbound. These cases represent the semantics of XQuery nested queries, behaving similar to a left outer join ($\exists\bowtie$).

$$
\begin{array}{l}
join_{nl}(\{\ \$Var_1, \cdots, \$Var_n\ \}, \$res) = \\
\left(\begin{array}{l} \texttt{xsp:isBlank}(\$Var_1) \texttt{ or} \\ \texttt{fn:empty}(\$res/\texttt{sr:binding}[\texttt{@name} = Var_1]/*) \texttt{ or} \\ (\$Var_1 \texttt{ eq } \$res/\texttt{sr:binding}[\texttt{@name} = Var_1]/*) \end{array} \right) \texttt{ and} \\
\qquad \cdots \\
\texttt{and } \left(\begin{array}{l} \texttt{xsp:isBlank}(\$Var_n) \texttt{ or} \\ \texttt{fn:empty}(\$res/\texttt{sr:binding}[\texttt{@name} = Var_n]/*) \texttt{ or} \\ (\$Var_n \texttt{ eq } \$res/\texttt{sr:binding}[\texttt{@name} = Var_n]/*) \end{array} \right)
\end{array}
$$

When Q is an XSPARQL expression of form

$$
\begin{array}{ll}
(1) & \texttt{for } \mathit{Vars}^{out}\ \mathit{DatasetClause}^{out}\ \mathit{WhereClause}^{out}\ \mathit{SolutionModifier}^{out} \\
(2) & \texttt{return} \\
(3) & \quad \texttt{for } \mathit{Vars}^{in}\ \mathit{DatasetClause}^{in}\ \mathit{WhereClause}^{in}\ \mathit{SolutionModifier}^{in} \\
(4) & \quad \texttt{return } \mathit{ExprSingle}
\end{array}
\tag{Q4}
$$

the application of the rewriting function $opt_{nl}(Q)$ can be split into two cases:

- in case *ExprSingle* does not contain any occurrences of (Q4) then, considering $\mathit{Vars}^{sp} = \mathit{vars}\big(\mathit{WhereClause}^{in}\big)$ the set of variables from the inner *WhereClause*, we have that:

$$
\begin{array}{ll}
 & opt_{nl}(Q) = \\
(1) & \texttt{let \$xsp:res\_in := xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars}^{in} \cup \mathit{Vars}^{out} \cap \mathit{Vars}^{sp} \\ \mathit{DatasetClause}^{in}\ \mathit{WhereClause}^{in} \\ \mathit{SolutionModifier}^{in}\end{array}\right)\texttt{ return} \\
(2) & \texttt{let \$xsp:res\_out := xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars}^{out}\ \mathit{DatasetClause}^{out} \\ \mathit{WhereClause}^{out}\ \mathit{SolutionModifier}^{out}\end{array}\right)\texttt{ return} \\
(3) & \texttt{for \$xsp:rout at \$xsp:posvar\_out in \$xsp:res\_out//sr:result return} \\
(4) & \quad \texttt{let } \$v := \texttt{\$xsp:rout/sr:binding[@name} = v\texttt{]/* return} \qquad \text{for each } \$v \in \mathit{Vars}^{out} \\
(5) & \quad \texttt{for \$xsp:rin at \$xsp:posvar\_out in \$xsp:res\_in//sr:result return} \\
(6) & \quad\quad \texttt{if } \Big(\mathit{join}_{sr}\Big(\ \mathit{Vars}^{out} \cap \mathit{Vars}^{sp}, \texttt{\$xsp:res\_out}, \texttt{\$xsp:res\_in}\ \Big)\Big)\texttt{ then} \\
(7) & \quad\quad\quad \texttt{let } \$v := \texttt{\$xsp:res\_in/sr:binding[@name} = v\texttt{]/* return} \qquad \text{for each } \$v \in \mathit{Vars}^{out} \triangle \mathit{Vars}^{sp} \\
(8) & \quad\quad\quad \mathit{ExprSingle} \\
(9) & \quad\quad \texttt{else } ()
\end{array}
$$

- otherwise:

$$
opt_{nl}(Q) = opt_{nl}\left(\begin{array}{l}\texttt{for } \mathit{Vars}^{out}\ \mathit{DatasetClause}^{out}\ \mathit{WhereClause}^{out}\ \mathit{SolutionModifier}^{out} \\ \texttt{return} \\ \quad \texttt{for } \mathit{Vars}^{in}\ \mathit{DatasetClause}^{in}\ \mathit{WhereClause}^{in}\ \mathit{SolutionModifier}^{in} \\ \quad \texttt{return } opt_{nl}(\mathit{ExprSingle})\end{array}\right)
$$

The $join_{sr}$ function is defined as:

$$
\begin{array}{c}
join_{sr}(\{\,\$\mathit{Var}_1, \cdots, \$\mathit{Var}_n\,\}, \$\mathit{resOut}, \$\mathit{resIn}) = \\
join_{nl}(\{\,\$\mathit{resOut}/\texttt{sr:binding[@name} = \mathit{Var}_1\texttt{]/*}\,\}, \$\mathit{resIn}) \\
\texttt{and } \cdots \texttt{ and} \\
join_{nl}(\{\,\$\mathit{resOut}/\texttt{sr:binding[@name} = \mathit{Var}_n\texttt{]/*}\,\}, \$\mathit{resIn})\ .
\end{array}
$$

The $join_{sr}$ function behaves in a similar fashion to the $join_{nl}$ function with the difference that it compares two SPARQL solution sequences. For nested expressions with an outer *SQLForClause*, i.e. when Q is an XSPARQL expression of form

$$
\begin{array}{ll}
(1) & \texttt{for } \mathit{AttrSpec}_1\ \texttt{as } \$\mathit{Var}_1, \ldots, \mathit{AttrSpec}_n\ \texttt{as } \$\mathit{Var}_n\ \mathit{RelationList}\ \mathit{SQLWhereClause} \\
(2) & \texttt{return} \\
(3) & \quad \texttt{for } \mathit{Vars}^{in}\ \mathit{DatasetClause}^{in}\ \mathit{WhereClause}^{in}\ \mathit{SolutionModifier}^{in} \\
(4) & \quad \texttt{return } \mathit{ExprSingle}
\end{array}
\tag{Q5}
$$

the application of the rewriting function $opt_{nl}(Q)$ can also be split into two cases:

- in case *ExprSingle* does not contain any occurrences of (Q5) then, considering $\mathit{Vars}^{sp} = \mathit{vars}\big(\mathit{WhereClause}^{in}\big)$ is the set of variables from the inner *WhereClause* and $\mathit{Vars}^{out} =$

$\{ \$Var_1, \ldots, \$Var_n \}$ is a shorthand notation for the variables in the outer *SQLForClause*, we have that:

$opt_{nl}(Q) =$

(1) `let $xsp:res_in :=` `xsp:sparqlCall` $\left(\begin{array}{l} \texttt{select } Vars^{in} \cup Vars^{out} \cap Vars^{sp} \\ DatasetClause^{in}\ \ WhereClause^{in} \\ SolutionModifier^{in} \end{array} \right)$ `return`

(2) `let $xsp:res_out :=` `xsp:sqlCall` $\left(\begin{array}{l} \texttt{select } AttrSpec_1, \ldots, AttrSpec_n \\ RelationList\ SQLWhereClause \end{array} \right)$ `return`

(3) `for $xsp:rout at $xsp:posvar_out in $xsp:res_out//sr:result return`

(4) `let` v `:= $xsp:rout/sr:binding[@name =` v`]/* return` for each $\$v \in Vars^{out}$

(5) `for $xsp:rin at $xsp:posvar_out in $xsp:res_in//sr:result return`

(6) `if` $\Big(join_{sr} \Big(Vars^{out} \cap Vars^{sp}, \$\texttt{xsp:res\_out}, \$\texttt{xsp:res\_in} \Big) \Big)$ `then`

(7) `let` $\$v :=$ `$xsp:res_in/sr:binding[@name =` v`]/* return` for each $\$v \in Vars^{out} \triangle Vars^{sp}$

(8) *ExprSingle*

(9) `else ()`

- otherwise:

$$opt_{nl}(Q) = opt_{nl}\left(\begin{array}{l} \texttt{for } AttrSpec_1 \texttt{ as } \$Var_1, \ldots, AttrSpec_n \texttt{ as } \$Var_n\ RelationList\ SQLWhereClause \\ \texttt{return} \\ \quad \texttt{for } Vars^{in}\ DatasetClause^{in}\ WhereClause^{in}\ SolutionModifier^{in} \\ \quad \texttt{return } opt_{nl}(ExprSingle) \end{array} \right)$$

The following proposition states that the opt_{nl} rewriting function is sound and complete.

Proposition 5.3. *Let Q be an XSPARQL expression of form* (Q3)*,* (Q4)*, or* (Q5) *and dynEnv the dynamic environment of Q, then $dynEnv \vdash Q \Rightarrow Val$ if and only if $dynEnv \vdash opt_{nl}(Q) \Rightarrow Val$.*

Proof: We now present the proof of the opt_{nl} rewriting function for expressions of the form (Q4). We start by showing the proof for the base case, where *ExprSingle* of (Q4) does not contain any occurrences of (Q4).

Base Case. ($\Rightarrow$) We start by showing that if $\text{dynEnv} \vdash Q \Rightarrow Val$ then $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow Val$. We present the proof tree for each of the XQuery core expressions in the $opt_{nl}(Q)$ rewriting where, in each proof tree, *Expr* corresponds to the XQuery expressions of the following lines.

`let` **expression of line (1).** For this rule let $Vars = Vars^{in} \cup \big(Vars^{out} \cap vars\big(WhereClause^{in} \big) \big)$ be the set of variables from the inner *SparqlForClause* and any variables from the outer *SparqlForClause* used in the inner *WhereClause*. Thus, we have that:

$$\frac{\text{dynEnv} \vdash \texttt{xsp:sparqlCall} \left(\begin{array}{l} \texttt{select } Vars\ DatasetClause^{in} \\ WhereClause^{in}\ SolutionModifier^{in} \end{array} \right) \Rightarrow \Omega_{nl}^{in}}{\text{dynEnv}_1^{nl} \vdash Expr \Rightarrow Res}$$

$$\text{dynEnv} \vdash \begin{array}{l} \texttt{let } \$\texttt{xsp:res\_in} := \texttt{xsp:sparqlCall} \left(\begin{array}{l} \texttt{select } Vars\ DatasetClause^{in} \\ WhereClause^{in}\ SolutionModifier^{in} \end{array} \right) \\ \quad \texttt{return } Expr \end{array} \Rightarrow Res$$

where

$$\text{dynEnv}_1^{nl} = \text{dynEnv} + \text{varValue}\big(\texttt{xsp:res\_in} \Rightarrow \Omega_{nl}^{in}\big) \quad . \tag{T3}$$

`let` expression of line (2):

$$\dfrac{\dfrac{\text{dynEnv}_1^{nl} \vdash \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars}^{out}\ \mathit{DatasetClause}^{out} \\ \mathit{WhereClause}^{out}\ \mathit{SolutionModifier}^{out}\end{array}\right) \Rightarrow \Omega_{nl}^{out}}{\text{dynEnv}_2^{nl} \vdash \mathit{Expr} \Rightarrow \mathit{Res}}}{\text{dynEnv}_1^{nl} \vdash \begin{array}{l}\texttt{let \$xsp:res\_out :=} \\ \quad \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars}^{out}\ \mathit{DatasetClause}^{out} \\ \mathit{WhereClause}^{out}\ \mathit{SolutionModifier}^{out}\end{array}\right) \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}}$$

where $\text{dynEnv}_2^{nl} = \text{dynEnv}_1^{nl} + \text{varValue}\big(\texttt{xsp:res\_out} \Rightarrow \Omega_{nl}^{out}\big)$.

`for` expression of line (3):

$$\dfrac{\dfrac{\text{dynEnv}_2^{nl} \vdash \texttt{\$xsp:res\_out//sr:result} \Rightarrow \mu_i^{out}}{\text{dynEnv}_3^{nl} \vdash \mathit{Expr} \Rightarrow \mathit{Res}_i} \quad \ddots}{\text{dynEnv}_2^{nl} \vdash \begin{array}{l}\texttt{for \$xsp:rout at \$xsp:posvar\_out} \\ \quad \texttt{in \$xsp:res\_out//sr:result} \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}_1, \cdots, \mathit{Res}_n}$$

where $\text{dynEnv}_3^{nl} = \text{dynEnv}_2^{nl} + \text{varValue}\left(\begin{array}{l}\texttt{xsp:rout} \Rightarrow \mu_i^{out}; \\ \texttt{xsp:posvar\_out} \Rightarrow i\end{array}\right)$.

`let` expressions of line (4). Here we consider all the `let` expressions represented by line (4), where $\$v \in \mathit{Vars}^{out}$:

$$\dfrac{\dfrac{\text{dynEnv}_3^{nl} \vdash \texttt{\$xsp:rout/sr:binding[@name} = v\texttt{]/*} \Rightarrow V}{\text{dynEnv}_4^{nl} \vdash \mathit{Expr} \Rightarrow \mathit{Res}}}{\text{dynEnv}_3^{nl} \vdash \begin{array}{l}\texttt{let \$}v := \texttt{ \$xsp:rout/sr:binding[@name} = v\texttt{]/*} \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}}$$

where $\text{dynEnv}_4^{nl} = \text{dynEnv}_3^{nl} + \text{varValue}(v \Rightarrow V)$.

`for` expression of line (5):

$$\dfrac{\dfrac{\text{dynEnv}_4^{nl} \vdash \texttt{\$xsp:res\_in//sr:result} \Rightarrow \mu_j^{out}}{\text{dynEnv}_5^{nl} \vdash \mathit{Expr} \Rightarrow \mathit{Res}_j} \quad \ddots}{\text{dynEnv}_4^{nl} \vdash \begin{array}{l}\texttt{for \$xsp:rin at \$xsp:posvar\_in} \\ \texttt{in \$xsp:res\_in//sr:result} \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}_1, \cdots, \mathit{Res}_n}$$

where $\text{dynEnv}_5^{nl} = \text{dynEnv}_4^{nl} + \text{varValue}\left(\begin{array}{l}\texttt{xsp:rin} \Rightarrow \mu_j^{out}; \\ \texttt{xsp:posvar\_in} \Rightarrow j\end{array}\right)$.

`if` expression of lines (6)–(9). In this rule, *Expr* represents the `let` expressions from lines (7)–(8):

$$\dfrac{\dfrac{\text{dynEnv}_5^{nl} \vdash \mathit{join}_{sr}\left(\begin{array}{l}\mathit{Vars}^{out} \cap \mathit{vars}(\mathit{WhereClause}), \\ \texttt{\$xsp:res\_out}, \texttt{\$xsp:res\_in}\end{array}\right) \Rightarrow \texttt{true}}{\text{dynEnv}_5^{nl} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Res}_1}}{\text{dynEnv}_5^{nl} \vdash \begin{array}{l}\texttt{if } \left(\mathit{join}_{sr}\left(\begin{array}{l}\mathit{Vars}^{out} \cap \mathit{vars}(\mathit{WhereClause}), \\ \texttt{\$xsp:res\_out}, \texttt{\$xsp:res\_in}\end{array}\right)\right) \\ \texttt{then } \mathit{Expr} \texttt{ else } ()\end{array} \Rightarrow \mathit{Res}_1}$$

`let` expressions of lines (7)–(8). Again, we consider all the `let` expressions represented by line (7), where $\$v \in Vars^{out} \triangle vars\big(WhereClause^{in}\big)$:

$$\dfrac{\dfrac{\text{dynEnv}_5^{nl} \vdash \$\texttt{xsp:res\_in/sr:binding}[\texttt{@name} = v]/* \Rightarrow V}{\text{dynEnv}_6^{nl} \vdash ExprSingle \Rightarrow Res}}{\text{dynEnv}_5^{nl} \vdash \begin{array}{l}\texttt{let } \$v := \ \$\texttt{xsp:res\_in/sr:binding}[\texttt{@name} = v]/* \\ \texttt{return } ExprSingle\end{array} \Rightarrow Res}$$

where $\text{dynEnv}_6^{nl} = \text{dynEnv}_5^{nl} + \text{varValue}(v \Rightarrow V)$.

Consider Ω_{xs}^{out} and Ω_{xs}^{in} the solution sequences returned by the evaluation of the outer and inner *SparqlForClause*s of Q, respectively, and the set of join variables $J = Vars^{out} \cap vars\big(WhereClause^{in}\big)$. Furthermore consider $\mu_{xs}^{out} \in \Omega_{xs}^{out}$ and $\mu_{xs}^{in} \in \Omega_{xs}^{in}$ the solution mappings that agree on the value of each join variable $j \in J$ from where *Val* is generated, i.e. there exists some dynamic environment dynEnv_i^{xs} based on dynEnv and extended with the variable mappings from μ_{xs}^{out} and μ_{xs}^{in} such that $\text{dynEnv}_i^{xs} \vdash ExprSingle \Rightarrow Val$.

Outer *SparqlForClause*: Regarding the *SparqlForClause* of lines (1)–(2) of Q (evaluated considering dynEnv), the $opt_{nl}(Q)$ translates it into the `xsp:sparqlCall` from line (2), which is evaluated over dynEnv_1^{nl}. Consider C_1 the expression context where dynEnv_1^{nl} is included, μ_{C_1} the XSPARQL instance mapping of C_1 and $P^{out} = \mu_{C_1}\big(WhereClause^{out}\big)$ the graph pattern obtained from replacing the variables in $WhereClause^{out}$ according to μ_{C_1}. From (T3) we can see that $dom(\mu_{C_1}) = dom(\mu_C) \cup \{\ \$\texttt{xsp:res\_in}\ \}$ but `$xsp:res_in` belongs to the `$xsp:` reserved namespace so it cannot be included in the variables of $WhereClause^{out}$ and we can observe that we obtain the same graph pattern P^{out} by replacing $WhereClause^{out}$ according to μ_C, i.e. $P^{out} = \mu_{C_1}\big(WhereClause^{out}\big) = \mu_C\big(WhereClause^{out}\big)$. Furthermore, let $\Omega_{xs}^{out} = eval_{xs}\big(DatasetClause^{out}, WhereClause^{out}, \mu_C\big)$ be the solution sequence resulting from evaluating the outer *SparqlForClause* according to XSPARQL semantics and $\Omega_{nl}^{out} = eval\big(DatasetClause^{out}, P^{out}\big)$ be the pattern solution resulting from evaluating the rewritten outer *SparqlForClause* according to SPARQL semantics. Following Lemma 5.1, we have that $\Omega_{xs}^{out} = \Omega_{nl}^{out} \bowtie \{\ \mu_C\ \}$ and, as we have seen from the proof of Proposition 5.2, since μ_C is already included in dynEnv, we have that $\Omega_{xs}^{out} = \Omega_{nl}^{out}$.

Inner *SparqlForClause*: The inner *SparqlForClause* from lines (3)–(4) of Q is evaluated considering some dynamic environment dynEnv_i^{xs} (with expression context C_i). On the other hand, the $opt_{nl}(Q)$ translates this inner expression into the `xsp:sparqlCall` of line (1), which is evaluated over the dynamic environment dynEnv (with expression context C). Consider μ_C the XSPARQL instance mapping of C and μ_{C_i} the XSPARQL instance mapping of C_i. Since dynEnv_i^{xs} is an extension of dynEnv we have that $dom(\mu_C) \subseteq dom(\mu_{C_i})$. Let $\Omega_{xs}^{in} = eval_{xs}\big(DatasetClause^{in}, WhereClause^{in}, \mu_{C_i}\big)$ be the solution sequence resulting from the evaluation of the inner *SparqlForClause* of Q and the solution sequence resulting from the evaluation of the `xsp:sparqlCall` function be $\Omega_{nl}^{in} = eval\big(DatasetClause^{in}, P^{in}\big)$, where $P^{in} = \mu_C(P)$ is the graph pattern obtained from replacing the variables in $WhereClause^{in}$ according to μ_C. As $dom(\mu_C) \subseteq dom(\mu_{C_i})$, i.e. μ_C contains less bindings for variables than μ_{C_i}, the rewritten graph pattern P_{in} contains more variables and we get that $\Omega_{xs}^{in} \preceq \Omega_{nl}^{in}$.

Since we know that $\Omega_{nl}^{out} = \Omega_{xs}^{out}$ and $\Omega_{xs}^{in} \preceq \Omega_{nl}^{in}$, we obtain that $\mu_{xs}^{out} \in \Omega_{nl}^{out}$ and $\mu_{xs}^{in} \in \Omega_{nl}^{in}$. Since $opt_{nl}(Q)$ performs a nested-loop iteration over Ω_{nl}^{out} and Ω_{nl}^{in}, the $join_{sr}$ function will join the two solution mappings successfully since μ_{xs}^{out} and μ_{xs}^{in} share the same values for the join variables, and thus we have that $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow Val$.

($\Leftarrow$) We now proceed by showing that if $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow Val$ then $\text{dynEnv} \vdash Q \Rightarrow Val$. Let us turn to the evaluation of $\text{dynEnv} \vdash Q \Rightarrow Val$.

***SparqlForClause* from lines (1)–(2).** Considering that *Expr* corresponds to the *SparqlForClause* from lines (3)–(4) of Q, the evaluation of this *SparqlForClause* consists of the application of Rule (D7):

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_m)}{\text{dynEnv} \vdash fs{:}dataset\big(DatasetClause^{out}\big) \Rightarrow DS^{out}} \quad \cdots}{\text{dynEnv} \vdash fs{:}sparql\left(\begin{array}{l} DS^{out},\ WhereClause, \\ SolutionModifier \end{array}\right) \Rightarrow \mu_i} \quad \cdots}{\text{dynEnv}_1^{xs} \vdash Expr \Rightarrow Value_i} \quad \cdots}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for}\ Vars^{out}\ DatasetClause^{out} \\ WhereClause^{out}\ SolutionModifier^{out} \\ \texttt{return}\ Expr \end{array} \Rightarrow Value_1, \cdots, Value_m}$$

with $Vars^{out} = \$Var_1^{out} \cdots \$Var_n^{out}$, we have for each μ_i

$$\text{dynEnv}_1^{xs} = \begin{array}{l} \text{dynEnv} + \text{activeDataset}\big(DS^{out}\big) + \text{globalPosition}((Pos_1, \cdots, Pos_m, i)) \\ + \text{varValue}\left(\begin{array}{l} Var_1^{out} \Rightarrow fs{:}value\big(\mu_i, Var_1^{out}\big)\,; \\ \ldots\,; \\ Var_n^{out} \Rightarrow fs{:}value\big(\mu_i, Var_n^{out}\big) \end{array}\right) \end{array} \tag{T4}$$

***SparqlForClause* of lines (3)–(4).** The evaluation of $\text{dynEnv}_1^{xs} \vdash Expr \Rightarrow Value_i$ is given by:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_m)}{\text{dynEnv}_1^{xs} \vdash fs{:}dataset\big(DatasetClause^{in}\big) \Rightarrow DS^{in}} \quad \cdots}{\text{dynEnv}_1^{xs} \vdash fs{:}sparql\left(\begin{array}{l} DS^{in}, WhereClause^{in}, \\ SolutionModifier^{in} \end{array}\right) \Rightarrow \mu_j} \quad \cdots}{\text{dynEnv}_2^{xs} \vdash ExprSingle \Rightarrow Value_j} \quad \cdots}{\text{dynEnv}_1^{xs} \vdash \begin{array}{l} \texttt{for}\ Vars^{in}\ DatasetClause^{in} \\ WhereClause^{in}\ SolutionModifier^{in} \\ \texttt{return}\ ExprSingle \end{array} \Rightarrow Value_1 \cdots Value_m}$$

where, considering $Vars^{in} = \$\,Var_1^{in} \ldots \$\,Var_n^{in}$, we have for each μ_j

$$\text{dynEnv}_2^{xs} = \begin{array}{l} \text{dynEnv}_1^{xs} + \text{activeDataset}\big(DS^{in}\big) + \text{globalPosition}((Pos_1, \cdots, Pos_m, j)) \\ + \text{varValue}\left(\begin{array}{l} Var_1^{in} \Rightarrow fs{:}value\big(\mu_j, Var_1^{in}\big)\,; \\ \cdots\,; \\ Var_n^{in} \Rightarrow fs{:}value\big(\mu_j, Var_n^{in}\big) \end{array}\right) \end{array}.$$

Let Ω_{nl}^{out} and Ω_{nl}^{in} be the pattern solutions returned by the outer and inner *SparqlForClause*s, respectively, and let $\mu_{nl}^{out} \in \Omega_{nl}^{out}$ and $\mu_{nl}^{in} \in \Omega_{nl}^{in}$ be the solution mappings. Without loss of generality we can assume these are the solution mappings from where *Val* is deduced, i.e. μ_{nl}^{out} and μ_{nl}^{in} are compatible. We also know that there exists a dynamic environment dynEnv^{nl}, based on dynEnv and extended with the variable mappings μ_{nl}^{out} and μ_{nl}^{in} such that $\text{dynEnv}^{nl} \vdash ExprSingle \Rightarrow Val$.

As we know from the ($\Rightarrow$) direction of the proof, $\Omega_{nl}^{out} = \Omega_{xs}^{out}$ and so we have that $\mu_{nl}^{out} \in \Omega_{xs}^{out}$. Regarding the evaluation of the inner *SparqlForClause* we also know that $\Omega_{xs}^{in} \preceq \Omega_{nl}^{in}$ and as such, we must consider two cases: (i) $\mu_{nl}^{in} \in \Omega_{xs}^{in}$ or (ii) $\mu_{nl}^{in} \notin \Omega_{xs}^{in}$. From (i), we immediately get the desired result that $\text{dynEnv} \vdash Q \Rightarrow Val$. For (ii), we know from (T4) that the inner *SparqlForClause* is executed over dynEnv_1^{xs} (and the respective XSPARQL instance mapping $\mu_{C_1}^{xs}$), which include the bindings for variables from each solution mapping $\mu_i \in \Omega_{xs}^{out}$. Thus, according to the XSPARQL BGP matching (cf. Definition 4.11), Ω_{xs}^{in} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega_{xs}^{out}$ and, since $\mu_{nl}^{out} \in \Omega_{xs}^{out}$, specifically those compatible with μ_{nl}^{out}. However, we know

that μ_{nl}^{in} is compatible with μ_{nl}^{out} and thus we have that μ_{nl}^{in} must also belong to Ω_{xs}^{in} and we can deduce that dynEnv $\vdash Q \Rightarrow Val$.

Inductive Step. The proof follows from the recursive application of the base case, over a new dynamic environment determined by the opt_{nl} rewriting to dynEnv$_i \vdash opt_{nl}(ExprSingle)$.

The proof for nested queries with an XQuery `for` outer expression (Q3) is analogous where, in the preceding, the evaluation of the *SparqlForClause* from lines (1)–(2) of (Q4) is replaced by the evaluation of an XQuery *ForClause*, as presented by (Draper, Fankhauser et al., 2010, Section 4.8.2). □

5.3.2. Dependent Join implementation in SPARQL

This form of rewriting of nested expressions aims at improving the runtime of the query by delegating the execution of the join to the SPARQL engine, as opposed to performing the join within XQuery (as in the previous optimisation). We start by presenting the rewriting function for the case when both nested expressions are *SparqlForClause*s: for such nested expressions we can implement the join by rewriting the *SparqlForClause*s into a single SPARQL query.

SparqlForClause within a *SparqlForClause*

The idea with these rewritings is that nested *SparqlForClause*s in XSPARQL can be implemented by a SPARQL query that merges the `where` clauses of the outer and inner *SparqlForClause*. However, there are some restrictions to the applicability of this rewriting: (i) both queries must be done over the same dataset; (ii) apart from `order by`, no other solution modifiers can be used in the queries; and (iii) the original queries must not require any nesting of the XML output or use of aggregators. The use of aggregators is restricted since in SPARQL queries they are only possible in the not yet standardised SPARQL 1.1. Thus it is not possible to generate the nested XML structure required by some queries, for example the query presented in Figure 5.3, by using a single SPARQL query or alternatively further processing of the SPARQL results in XQuery. As indicated before, for the next rewriting we are only allowing the `order by` solution modifier and the concatenation of "`order by` $o1" and "`order by` $o2" is "`order by` $o1 $o2".

For an XSPARQL query Q of form:[13]

$$
\begin{array}{ll}
(1) & \texttt{for } Vars^{out}\ DatasetClause\ \texttt{where } GGP^{out}\ \texttt{order by } OC^{out} \\
(2) & \texttt{return} \\
(3) & \quad \texttt{for } Vars^{in}\ DatasetClause\ \texttt{where } GGP^{in}\ \texttt{order by } OC^{in} \\
(4) & \quad \texttt{return } ExprSingle
\end{array}
\tag{Q6}
$$

then

- in case *ExprSingle* does not contain any occurrences of (Q6), we have that:

$$
\begin{array}{ll}
 & opt_{sr}(Q) = \\
(1) & \texttt{let \$xsp:results := xsp:sparqlCall} \left(\begin{array}{l} \texttt{select } Vars^{out} \cup Vars^{in}\ DatasetClause \\ \texttt{where } \{\ GGP^{out}\ .\ GGP^{in}\ \} \\ \texttt{order by } OC^{out}\ OC^{in} \end{array} \right) \texttt{return} \\
(2) & \texttt{for \$xsp:result at \$xsp:posvar in \$xsp:results//sr:result return} \\
(3) & \quad \texttt{let } \$v := \texttt{\$xsp:result/sr:binding[@name} = \$v\texttt{]/*} \texttt{ return} \qquad \text{for each } \$v \in Vars^{out} \cup Vars^{in} \\
(4) & \quad ExprSingle
\end{array}
$$

[13] For presentation purposes, *GGP* and *OC* are a short representation for *GroupGraphPattern* and *OrderCondition*, respectively.

Please note that the group graph patterns GGP_1 and GGP_2 include the surrounding curly braces: { and }.

- otherwise:

$$opt_{sr}(Q) = opt_{sr}\left(\begin{array}{l}\texttt{for } Vars^{out}\ DatasetClause \texttt{ where } GGP^{out} \texttt{ order by } OC^{out} \\ \texttt{return} \\ \quad \texttt{for } Vars^{in}\ DatasetClause \texttt{ where } GGP^{in} \texttt{ order by } OC^{in} \\ \quad \texttt{return } opt_{sr}(ExprSingle)\end{array}\right)$$

Proposition 5.4. *Let Q an XSPARQL expression of form* (Q6) *and dynEnv the dynamic environment of Q, then dynEnv $\vdash Q \Rightarrow Val$ if and only if dynEnv $\vdash opt_{sr}(Q) \Rightarrow Val$.*

Proof: We start by showing the proof for the base case, where *ExprSingle* of (Q6) does not contain any occurrences of (Q6).

Base Case. ($\Rightarrow$) We start by showing that if dynEnv $\vdash Q \Rightarrow Val$ then dynEnv $\vdash opt_{sr}(Q) \Rightarrow Val$. Next, we show the proof tree for each of the XQuery core expressions in each line of the opt_{sr} rewriting where, for each line, *Expr* represents the expressions of the following lines.

`let` **expression of line (1):**

$$\dfrac{\text{dynEnv} \vdash \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } Vars^{out} \cup Vars^{in} \\ DatasetClause \\ \texttt{where } \{\ GGP^{out}\ .\ GGP^{in}\ \} \\ \texttt{order by } OC^{out}\ OC^{in}\end{array}\right) \Rightarrow \Omega_{sr}}{\text{dynEnv}_1^{sr} \vdash Expr \Rightarrow Res}$$

$$\text{dynEnv} \vdash \begin{array}{l}\texttt{let \$xsp:results :=} \\ \quad \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } Vars^{out} \cup Vars^{in}\ DatasetClause \\ \texttt{where } \{\ GGP^{out}\ .\ GGP^{in}\ \} \\ \texttt{order by } OC^{out}\ OC^{in}\end{array}\right) \\ \texttt{return } Expr\end{array} \Rightarrow Res$$

where $\text{dynEnv}_1^{sr} = \text{dynEnv} + \text{varValue}(\texttt{xsp:results} \Rightarrow \Omega_{sr})$.

`for` **expression of line (2):**

$$\dfrac{\dfrac{\text{dynEnv}_1^{sr} \vdash \texttt{\$xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}_2^{sr} \vdash ExprSingle \Rightarrow Res_i} \quad \cdots}{\text{dynEnv}_1^{sr} \vdash \begin{array}{l}\texttt{for \$xsp:result at \$xsp:posvar} \\ \quad \texttt{in \$xsp:results//sr:result} \\ \texttt{return } ExprSingle\end{array} \Rightarrow Res_1, \cdots, Res_n}$$

where $\text{dynEnv}_2^{sr} = \text{dynEnv}_1^{sr} + \text{varValue}\left(\begin{array}{l}\texttt{xsp:result} \Rightarrow \mu_i; \\ \texttt{xsp:posvar} \Rightarrow i\end{array}\right)$.

`let` **expressions of lines (3)–(4).** Here we consider all the `let` expressions represented by line (3), where $\$v \in Vars^{out} \cup Vars^{in}$:

$$\dfrac{\dfrac{\text{dynEnv}_2^{sr} \vdash \texttt{\$xsp:result/sr:binding[@name} = \$v\texttt{]/*} \Rightarrow V}{\text{dynEnv}_3^{sr} \vdash ExprSingle \Rightarrow Res}}{\text{dynEnv}_2^{sr} \vdash \begin{array}{l}\texttt{let } \$v := \texttt{\$xsp:result/sr:binding[@name} = v\texttt{]/*} \\ \texttt{return } ExprSingle\end{array} \Rightarrow Res}$$

where $\text{dynEnv}_3^{sr} = \text{dynEnv}_2^{sr} + \text{varValue}(v \Rightarrow V)$.

Let Ω_{xs}^{out} and Ω_{xs}^{in} be the solution sequences returned by the evaluation of the outer and inner *SparqlForClause*s of Q, respectively. Furthermore, let $\mu_{xs}^{out} \in \Omega_{xs}^{out}$ and $\mu_{xs}^{in} \in \Omega_{xs}^{in}$ be compatible solution mappings and dynEnv_i^{expr} the dynamic environment that results from extending dynEnv with the variable mappings from μ_{xs}^{out} and μ_{xs}^{in}, such that $\text{dynEnv}_i^{expr} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Val}$.

According to the SPARQL semantics, the solution sequence that results from evaluating the graph pattern "$\{\ GGP^{out}\ .\ GGP^{in}\ \}$", $\Omega_{sr} = \Omega_{sr}^{out} \bowtie \Omega_{sr}^{in}$ consists of all the solution mappings $\mu_{sr}^{out} \in \Omega_{sr}^{out}$ and $\mu_{sr}^{in} \in \Omega_{sr}^{in}$ such that μ_{sr}^{out} and μ_{sr}^{in} are *compatible*. The evaluation of the outer *SparqlForClause* (lines (1)–(2) of Q), evaluated over dynEnv, is translated by $opt_{sr}(Q)$ into the `xsp:sparqlCall` from line (1), which is also evaluated over dynEnv. In this case, according to Lemma 5.1, we have that $\Omega_{sr}^{out} = \Omega_{xs}^{out}$ and then $\mu_{xs}^{out} \in \Omega_{sr}^{out}$.

The inner *SparqlForClause* (lines (3)–(4) of Q), which is evaluated over some dynamic environment dynEnv_i^{xs}, is incorporated by the $opt_{sr}(Q)$ rewriting into the `xsp:sparqlCall` from line (1), which is also evaluated over dynEnv. Considering that dynEnv is less restrictive than dynEnv_i^{xs}, i.e. dynEnv contains less bindings for variables than dynEnv_i^{xs}, and thus the evaluation of the inner *SparqlForClause* over dynEnv will contain all the solution mappings from Ω_{xs}^{in} and specifically μ_{xs}^{in}. As μ_{xs}^{out} and μ_{xs}^{in} are *compatible* we have that $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow \mathit{Val}$.

Please note that we are only considering `order by` solution modifiers, thus the number of results of each query is not modified. The ordering of the results may be changed but this does not interfere with this proof and solution modifiers can be safely ignored.

($\Leftarrow$) Next we show that if $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow \mathit{Val}$ then $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$. Let us turn to the evaluation of $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$.

***SparqlForClause* from lines (1)–(2).** Where *Expr* corresponds to the *SparqlForClause* from lines (3)–(4) of Q. The evaluation of this *SparqlForClause* consists of the application of Rule (D7):

$$\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (\mathit{Pos}_1, \cdots, \mathit{Pos}_m)}{\text{dynEnv} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{DS}}}{\text{dynEnv} \vdash \mathit{fs{:}sparql}\left(\begin{array}{l}\mathit{DS},\ \texttt{where}\ \mathit{GGP}^{out}\\ \texttt{order by}\ \mathit{OC}^{out}\end{array}\right) \Rightarrow \mu_i}}{\text{dynEnv}_1^{xs} \vdash \mathit{Expr} \Rightarrow \mathit{Value}_i} \quad \cdots}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for}\ \mathit{Vars}^{out}\ \mathit{DatasetClause}\\ \texttt{where}\ \mathit{GGP}^{out}\ \texttt{order by}\ \mathit{OC}^{out}\ \Rightarrow \mathit{Value}_1, \cdots, \mathit{Value}_m\\ \texttt{return}\ \mathit{Expr}\end{array}}$$

where $\mathit{Vars}^{out} = \$\mathit{Var}_1^{out} \cdots \$\mathit{Var}_n^{out}$, we have for each μ_i

$$\text{dynEnv}_1^{xs} = \begin{array}{l}\text{dynEnv} + \text{activeDataset}(\mathit{DS}) + \text{globalPosition}((\mathit{Pos}_1, \cdots, \mathit{Pos}_m, i))\\ + \text{varValue}\left(\begin{array}{l}\mathit{Var}_1^{out} \Rightarrow \mathit{fs{:}value}(\mu_i, \mathit{Var}_1^{out})\ ;\\ \cdots\ ;\\ \mathit{Var}_n^{out} \Rightarrow \mathit{fs{:}value}(\mu_i, \mathit{Var}_n^{out})\end{array}\right)\end{array} . \qquad \text{(T5)}$$

***SparqlForClause* of lines (3)–(4).** The evaluation of $\text{dynEnv}_i^{xs} \vdash \mathit{ExprSingle}^{out} \Rightarrow \mathit{Value}_i$ is shown next:

$$\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (\mathit{Pos}_1, \cdots, \mathit{Pos}_m)}{\text{dynEnv}_1^{xs} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{DS}}}{\text{dynEnv}_1^{xs} \vdash \mathit{fs{:}sparql}\left(\begin{array}{l}\mathit{DS},\ \texttt{where}\ \mathit{GGP}^{in}\\ \texttt{order by}\ \mathit{OC}^{in}\end{array}\right) \Rightarrow \mu_j}}{\text{dynEnv}_2^{xs} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_j} \quad \cdots}{\text{dynEnv}_1^{xs} \vdash \begin{array}{l}\texttt{for}\ \mathit{Vars}^{in}\ \mathit{DatasetClause}\\ \texttt{where}\ \mathit{GGP}^{in}\ \texttt{order by}\ \mathit{OC}^{in}\ \Rightarrow \mathit{Value}_1, \cdots, \mathit{Value}_m\\ \texttt{return}\ \mathit{ExprSingle}\end{array}}$$

where $Vars^{in} = \$Var_1^{in} \cdots \$Var_n^{in}$, for each μ_j we have that:

$$\text{dynEnv}_2^{xs} = \begin{array}{l} \text{dynEnv}_1^{xs} + \text{activeDataset}(DS) + \text{globalPosition}((Pos_1, \cdots, Pos_m, j)) \\ + \text{varValue}\left(\begin{array}{l} Var_1^{in} \Rightarrow fs{:}value\big(\mu_j, Var_1^{in}\big)\,; \\ \cdots\,; \\ Var_n^{in} \Rightarrow fs{:}value\big(\mu_j, Var_n^{in}\big) \end{array}\right) \end{array}.$$

As we have seen in the ($\Rightarrow$) direction, we have that $\Omega_{sr}^{out} = \Omega_{xs}^{out}$ and so we have that $\mu_{sr}^{out} \in \Omega_{xs}^{out}$. Furthermore let Ω_{sr}^{out} and Ω_{sr}^{in} be as per the ($\Rightarrow$) direction of the proof. As we have seen, Ω_{sr} contains all the solution mappings $\mu = \mu_{sr}^{out} \bowtie \mu_{sr}^{in}$ such that $\mu_{sr}^{out} \in \Omega_{sr}^{out}$ and $\mu_{sr}^{in} \in \Omega_{sr}^{in}$ and μ_{sr}^{out} and μ_{sr}^{in} are *compatible*. Without loss of generality let us consider μ_{sr}^{out} and μ_{sr}^{in} the solution mappings where *Val* is deduced from.

Let C be the expression context where dynEnv is included and μ_C the XSPARQL instance mapping of C. Furthermore, let $P^{in} = \mu_C\big(GGP^{in}\big)$ be the graph pattern obtained from replacing the variables in GGP^{in} according to μ_C. Since $vars\big(GGP^{in}\big) \subseteq vars\big(P^{in}\big)$ all solutions mappings returned by evaluating GGP^{in} under XSPARQL semantics are included in the solution sequence of evaluating P^{in} under SPARQL semantics i.e. $\Omega_{xs}^{in} \preceq \Omega_{sr}^{in}$. We obtain two cases: (i) $\mu_{sr}^{in} \in \Omega_{xs}^{in}$ or (ii) $\mu_{sr}^{in} \notin \Omega_{xs}^{in}$. From (i) we immediately get that $\text{dynEnv} \vdash Q \Rightarrow Val$. For (ii), consider $\mu_{C_1}^{xs}$ the XSPARQL instance of the inner *SparqlForClause* (created based on dynEnv_1^{xs}). As we can see from (T5), dynEnv_1^{xs} (and thus also $\mu_{C_1}^{xs}$) includes the bindings for variables from each solution mapping $\mu_i \in \Omega_{xs}^{out}$. Thus, according to the XSPARQL BGP matching (cf. Definition 4.11), Ω_{xs}^{in} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega_{xs}^{out}$ and specifically those compatible with μ_{sr}^{out}. Since we know that μ_{sr}^{in} is compatible with μ_{sr}^{out}, we have that μ_{sr}^{in} must belong to Ω_{xs}^{in}, thus we can deduce that $\text{dynEnv} \vdash Q \Rightarrow Val$.

Inductive Step. The proof follows from the recursive application of the base case, over a new dynamic environment determined by the opt_{sr} rewriting to $\text{dynEnv}_i \vdash opt_{sr}(ExprSingle)$. □

SparqlForClause within an XQuery `for`

In case the outer expression is an XQuery `for` or an XSPARQL *SQLForClause* a similar strategy of deferring the join to a single SPARQL query is still possible. This optimisation relies on first transforming the outer expressions' XML results into RDF and then joining this newly created RDF graph with the inner *SparqlForClause*'s `where` pattern in a single SPARQL query. For the implementation of this optimisation we can rely on a triple store with support for named graphs and temporarily store the bindings for dependent variables from the outer XQuery `for` expression's as RDF triples. We can then execute a combined query with an adapted graph pattern, that joins the pattern in the `where` clause of the inner *SparqlForClause* with the bindings stored in the newly created named graph. The opt_{ng} rewriting function (presented below) starts by creating RDF triples representing the XML input, which are then collected into the variable `$xsp:ds` corresponding to the RDF graph to be inserted into the triple store. This operation is achieved by the XSPARQL functions `xsp:createNG` that returns a URI for the newly inserted RDF named graph, which is distinct from any other URIs for named graphs used in the query or present in the triple store, while finally the function `xsp:deleteNG` takes care of deleting the temporary graph. We will show this optimisation only for the case where the outer expression is an XQuery `for`, the case of an outer XSPARQL *SQLForClause* expression is analogous. Let Q be an XSPARQL expression of form

(1) `for` $\$VarName\ OptTypeDeclaration\ OptPositionalVar$ `in` $ExprSingle_1$
(2) `return`
(3) `for` $Vars\ DatasetClause\ WhereClause\ SolutionModifier$
(4) `return` $ExprSingle_2$ (Q7)

then

- in case $ExprSingle_1$ and $ExprSingle_2$ do not contain any occurrences of (Q7), we have that:

$opt_{ng}(Q) =$

(1) `let $xsp:ds :=` `xsp:createNG` $\left(\begin{array}{l} \texttt{for}\ \$VarName\ OptTypeDeclaration \\ OptPositionalVar\ \texttt{in}\ ExprSingle_1 \\ \texttt{return}\ \texttt{xsp:evalCT}(NGP) \end{array}\right)$ `return`
(2) `let $xsp:results :=`
`xsp:sparqlCall` $\left(\begin{array}{l} \texttt{select}\ Vars \cup \{\ \$VarName\ \} \\ DatasetClause\ \cup \{\ \texttt{from named \$xsp:ds}\ \} \\ WhereClause\ \cup \texttt{where}\{\ \texttt{graph \$xsp:ds}\ NGP\ \} \\ SolutionModifier \end{array}\right)$ `return`
(3) `for $xsp:result at $xsp:result_pos in $xsp:results//sr:result return`
(4) `let` $\$v :=$ `$xsp:result/sr:binding[@name =` $\$v$`]/*` for each $\$v \in Vars \cup \{\ \$VarName\ \}$
(5) `return` $(ExprSingle_2,$ `xsp:deleteNG($xsp:ds)`$)$

where NGP is the graph pattern { [] :`value` $\$VarName$ }.

- otherwise:

$$opt_{ng}(Q) = opt_{ng}\left(\begin{array}{l} \texttt{for}\ \$VarName\ OptTypeDeclaration\ OptPositionalVar\ \texttt{in}\ opt_{ng}(ExprSingle_1) \\ \texttt{return} \\ \quad \texttt{for}\ Vars\ DatasetClause\ WhereClause\ SolutionModifier \\ \quad \texttt{return}\ opt_{ng}(ExprSingle_2) \end{array}\right)$$

Let Q be an XSPARQL expression of form

(1) `for` $AttrSpec_1$ `as` $\$Var_1, \ldots,$ $AttrSpec_n$ `as` $\$Var_n\ RelationList\ SQLWhereClause$
(2) `return`
(3) `for` $Vars\ DatasetClause\ WhereClause\ SolutionModifier$
(4) `return` $ExprSingle$ (Q8)

then

- in case $ExprSingle$ does not contain any occurrences of (Q8), we have that:

$opt_{ng}(Q) =$

(1) `let $xsp:ds :=`
`xsp:createNG` $\left(tr\left(\begin{array}{l} \texttt{for}\ AttrSpec_1\ \texttt{as}\ \$Var_1, \ldots, AttrSpec_n\ \texttt{as}\ \$Var_n \\ RelationList\ SQLWhereClause \\ \texttt{return}\ \texttt{xsp:evalCT}(NGP) \end{array}\right)\right)$ `return`

(2) `let $xsp:results :=`

$$\texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars} \cup \{\, \$\mathit{Var}_1, \ldots, \$\mathit{Var}_n \,\} \\ \mathit{DatasetClause} \cup \{\, \texttt{from named \$xsp:ds} \,\} \\ \mathit{WhereClause} \cup \texttt{where} \{\, \texttt{graph \$xsp:ds } \mathit{NGP} \,\} \\ \mathit{SolutionModifier}\end{array}\right) \texttt{ return}$$

(3) `for $xsp:result at $xsp:result_pos in $xsp:results//sr:result return`

(4) `let` $\$v :=$ `$xsp:result/sr:binding[@name =` $\$v$`]/*` for each $\$v \in \mathit{Vars} \cup \{\, \$\mathit{Var}_1, \ldots, \$\mathit{Var}_n \,\}$

(5) `return` $(\mathit{ExprSingle}, \texttt{xsp:deleteNG(\$xsp:ds)})$

where NGP is the graph pattern $\{\, [] :\mathit{Var}_1\ \$\mathit{Var}_1;\ \ldots;\ :\mathit{Var}_n\ \$\mathit{Var}_n \,\}$.

- otherwise:

$$\mathit{opt}_{ng}(Q) = \mathit{opt}_{ng}\left(\begin{array}{l}\texttt{for } \mathit{AttrSpec}_1 \texttt{ as } \$\mathit{Var}_1, \ldots, \mathit{AttrSpec}_n \texttt{ as } \$\mathit{Var}_n\ \mathit{RelationList}\ \mathit{SQLWhereClause} \\ \texttt{return} \\ \quad \texttt{for } \mathit{Vars}\ \mathit{DatasetClause}\ \mathit{WhereClause}\ \mathit{SolutionModifier} \\ \quad \texttt{return } \mathit{opt}_{ng}(\mathit{ExprSingle})\end{array}\right)$$

Proposition 5.5. *Let Q be an XSPARQL expression of form* (Q7) *or* (Q8) *and dynEnv the dynamic environment of Q, then $\mathit{dynEnv} \vdash Q \Rightarrow \mathit{Val}$ if and only if $\mathit{dynEnv} \vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$.*

Proof: We start by showing the proof for the base case, where $\mathit{ExprSingle}_1$ and $\mathit{ExprSingle}_2$ of (Q7) do not contain any occurrences of (Q7).

Base Case. ($\Rightarrow$) Let us start by showing that if $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$ then $\text{dynEnv} \vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$. We now show the proof tree for each of the XQuery core expressions in the opt_{ng} rewriting.

`let` **expression of line (1).** Considering $NGP = \{\, [] :\texttt{value } \$\mathit{VarName} \,\}$, we have

$$\frac{\text{dynEnv} \vdash \texttt{xsp:createNG}\left(\begin{array}{l}\texttt{for } \$\mathit{VarName}\ \mathit{OptTypeDeclaration} \\ \mathit{OptPositionalVar} \texttt{ in } \mathit{ExprSingle}_1 \\ \texttt{return xsp:evalTemplate}(\mathit{NGP})\end{array}\right) \Rightarrow \mathit{DS}}{\text{dynEnv}_1^{ng} \vdash \mathit{Expr} \Rightarrow \mathit{Res}}$$

$$\text{dynEnv} \vdash \begin{array}{l}\texttt{let \$xsp:ds :=} \\ \quad \texttt{xsp:createNG}\left(\begin{array}{l}\texttt{for } \$\mathit{VarName}\ \mathit{OptTypeDeclaration} \\ \mathit{OptPositionalVar} \texttt{ in } \mathit{ExprSingle}_1 \\ \texttt{return xsp:evalTemplate}(\mathit{NGP})\end{array}\right) \\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}$$

where

$$\text{dynEnv}_1^{ng} = \text{dynEnv} \ + \text{varValue}(\texttt{xsp:ds} \Rightarrow \mathit{DS}) \quad . \tag{T6}$$

`let` **expression of line (2).** As a shortcut representation, consider the dataset clause $\mathit{DatasetClause}^{ng} = \mathit{DatasetClause} \cup \{\, \texttt{from named \$xsp:ds} \,\}$ and the graph pattern $\mathit{WhereClause}^{ng} = \mathit{WhereClause} \cup \texttt{where} \{\, \texttt{graph \$xsp:ds} \{\, [] :\texttt{value } \$\mathit{VarName} \,\} \,\}$.

$$\frac{\text{dynEnv}_1^{ng} \vdash \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars} \cup \{\,\$\mathit{VarName}\,\}\\ \mathit{DatasetClause}^{ng}\ \ \mathit{WhereClause}^{ng}\\ \mathit{SolutionModifier}\end{array}\right) \Rightarrow \Omega_{ng}}{\text{dynEnv}_2^{ng} \vdash \mathit{Expr} \Rightarrow \mathit{Res}}$$

$$\text{dynEnv}_1^{ng} \vdash \begin{array}{l}\texttt{let \$xsp:results :=}\\ \quad\texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select } \mathit{Vars} \cup \{\,\$\mathit{VarName}\,\}\\ \mathit{DatasetClause}^{ng}\ \ \mathit{WhereClause}^{ng}\\ \mathit{SolutionModifier}\end{array}\right)\\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}$$

where $\text{dynEnv}_2^{ng} = \text{dynEnv}_1^{ng} + \text{varValue}\big(\texttt{xsp:results} \Rightarrow \Omega_{ng}\big)$.

`for` expression of line (3):

$$\frac{\dfrac{\text{dynEnv}_2^{ng} \vdash \$\texttt{xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}_3^{ng} \vdash \mathit{Expr} \Rightarrow \mathit{Res}_i} \quad \cdots}{\text{dynEnv}_2^{ng} \vdash \begin{array}{l}\texttt{for \$xsp:result at \$xsp:result\_pos}\\ \quad\texttt{in \$xsp:results//sr:result}\\ \texttt{return } \mathit{Expr}\end{array} \Rightarrow \mathit{Res}_1, \cdots, \mathit{Res}_n}$$

where $\text{dynEnv}_3^{ng} = \text{dynEnv}_2^{ng} + \text{varValue}\left(\begin{array}{l}\texttt{xsp:result} \Rightarrow \mu_i;\\ \texttt{xsp:result\_pos} \Rightarrow i\end{array}\right)$.

`let` expressions of lines (4)–(5). Here we consider all the `let` expressions represented by line (4), where $\$v \in \mathit{Vars}$:

$$\frac{\dfrac{\text{dynEnv}_3^{ng} \vdash \$\texttt{xsp:result/sr:binding[@name} = \$v\texttt{]/*} \Rightarrow V}{\text{dynEnv}_4^{ng} \vdash \mathit{Expr} \Rightarrow \mathit{Res}}}{\text{dynEnv}_3^{ng} \vdash \begin{array}{l}\texttt{let } \$v := \$\texttt{xsp:result/sr:binding[@name} = \$v\texttt{]/*}\\ \texttt{return } \mathit{ExprSingle}_2\end{array} \Rightarrow \mathit{Res}}$$

where $\text{dynEnv}_4^{ng} = \text{dynEnv}_3^{ng} + \text{varValue}(v \Rightarrow V)$.

Let Ω_{xs}^{in} be the solution sequence returned by the evaluation of the inner *SparqlForClause* of Q. Furthermore let dynEnv_i^{expr} be the dynamic environment such that $\text{dynEnv}_i^{expr} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Val}$. dynEnv_i^{expr} results from extending dynEnv with bindings for the outer variable $\$\mathit{VarName}$ and with variable bindings from a solution mapping $\mu_{xs}^{in} \in \Omega_{xs}^{in}$ where $\mu_{xs}^{in}(\mathit{VarName}) = \$\mathit{VarName}$, i.e. the value for the join variable in the solution mapping μ_{xs}^{in} is the same as assigned to $\$\mathit{VarName}$.

The new merged dataset, $\mathit{DatasetClause}^{ng}$, is created based on *DatasetClause* and the newly created named graph *NG*. Since the URI that identifies the newly created named graph *NG* is distinct from any URI of named graphs present in *DatasetClause*, the triples included in *NG* will never be a solution for *WhereClause*, and will be matched only by the graph pattern "`where { graph` $`xsp:ds { [] :value` $\mathit{VarName}$ `} }`".

Let C be the expression context where dynEnv is included, μ_C the XSPARQL instance mapping of C and P^{out} and P^{in} the graph patterns obtained from replacing the variables in *WhereClause* and "`where { graph` $`xsp:ds { [] :value` $\mathit{VarName}$ `} }`" according to μ_C, respectively.

Furthermore, let $\Omega_{ng}^{out} = \mathit{eval}(\mathit{DatasetClause}^{ng}, P^{out})$ and $\Omega_{ng}^{in} = \mathit{eval}\big(\mathit{DatasetClause}^{ng}, P^{in}\big)$. According to SPARQL semantics, the pattern solution that results from evaluating *WhereClause*, $\Omega_{ng} = \Omega_{ng}^{out} \bowtie \Omega_{ng}^{in}$ consists of all the solution mappings $\mu_{out} \in \Omega_{ng}^{out}$ and $\mu_{in} \in \Omega_{ng}^{in}$ such that μ_{out} and μ_{in} are *compatible*. Similar to the proof of Proposition 5.4, we are only considering `order by` solution modifiers, these only change the order of the solution sequences and thus can be safely ignored for this proof.

The evaluation of the outer XQuery `for` clause (lines (1)–(2) of Q) performed over dynEnv is translated, by the $opt_{ng}(Q)$ function, into the `xsp:sparqlCall` from line (2), which is evaluated over dynEnv_1^{ng}.

However, as we can see from (T6), dynEnv_1^{ng} is based on dynEnv by adding the value for the `xsp:ds` variable and, since this variable belongs to the `xsp:` reserved namespace, it is not allowed to appear in the *WhereClause* and we have that the results of evaluating the `xsp:sparqlCall` function over dynEnv or dynEnv_1^{ng} will be the same.

The inner *SparqlForClause* (lines (3)–(4) of Q) is evaluated over some dynamic environment dynEnv^{expr}, is incorporated by the $opt_{ng}(Q)$ into the `xsp:sparqlCall` from line (2), which is evaluated over dynEnv_1^{ng}. Considering that dynEnv_1^{ng} is less restrictive than dynEnv^{expr}, i.e. dynEnv_1^{ng} contains less bindings for variables than dynEnv^{expr}, the evaluation of the inner *SparqlForClause* over dynEnv_1^{ng} will contain all the solution mappings from Ω_{xs}^{in} and specifically μ_{in}. As μ_{out} and μ_{in} are *compatible* we have that $\text{dynEnv} \vdash ng(expr) \Rightarrow Val$.

($\Leftarrow$) Next we will show that if $\text{dynEnv} \vdash opt_{ng}(Q) \Rightarrow Val$ then $\text{dynEnv} \vdash Q \Rightarrow Val$. Let us turn to the evaluation of $\text{dynEnv} \vdash Q \Rightarrow Val$.

XQuery `for` clause from lines (1)–(2). Here *Expr* corresponds to the *SparqlForClause* from lines (3)–(4) of Q.

$$\frac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_m)}{\text{dynEnv} \vdash ExprSingle_1 \Rightarrow V_i}}{\text{dynEnv}_i^{xs} \vdash Expr \Rightarrow Value_i} \quad \cdots}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for}\ \$VarName\ OptTypeDeclaration \\ \quad OptPositionalVar\ \texttt{in}\ ExprSingle_1 \\ \texttt{return}\ Expr\end{array} \Rightarrow Value_i, \ldots, Value_n}$$

we have for each V_i:

$$\text{dynEnv}_i^{xs} = \text{dynEnv} + \text{globalPosition}((Pos_1, \cdots, Pos_m, i)) + \text{varValue}(VarName \Rightarrow V_i) \quad . \tag{T7}$$

***SparqlForClause* of lines (2)–(4):**

$$\frac{\dfrac{\dfrac{\dfrac{\text{dynEnv.globalPosition} = (Pos_1, \cdots, Pos_m)}{\text{dynEnv}_i^{xs} \vdash fs{:}dataset(DatasetClause) \Rightarrow DS}}{\text{dynEnv}_i^{xs} \vdash fs{:}sparql\left(\begin{array}{l}DS, WhereClause, \\ SolutionModifier\end{array}\right) \Rightarrow \mu_j} }{\text{dynEnv}_j^{xs} \vdash ExprSingle_2 \Rightarrow Value_j} \quad \cdots}{\text{dynEnv}_i^{xs} \vdash \begin{array}{l}\texttt{for}\ Vars\ DatasetClause \\ \quad WhereClause\ SolutionModifier \\ \texttt{return}\ ExprSingle_2\end{array} \Rightarrow Value_1 \cdots Value_m}$$

where, considering $Vars = \$Var_1 \ldots \$Var_n$, we have for each μ_j:

$$\text{dynEnv}_j^{xs} = \begin{array}{l}\text{dynEnv}_i^{xs} + \text{activeDataset}(DS) + \text{globalPosition}((Pos_1, \cdots, Pos_m, j)) \\ \quad + \text{varValue}\left(\begin{array}{l}Var_1 \Rightarrow fs{:}value(\mu_j, Var_1)\,; \\ \cdots\,; \\ Var_n \Rightarrow fs{:}value(\mu_j, Var_n)\end{array}\right)\end{array} \quad .$$

As we have seen in the ($\Rightarrow$) direction, we have that $\Omega_{ng}^{out} = \Omega_{xs}^{out}$ and so we have that $\mu_{ng}^{out} \in \Omega_{xs}^{out}$. Let Ω_{ng}^{out} and Ω_{ng}^{in} be the solution sequences returned by the evaluation of the new $WhereClause^{ng}$ and *WhereClause*, respectively. As we have seen Ω_{ng} contains all the solution mappings $\mu = \mu_{ng}^{out} \bowtie \mu_{ng}^{in}$, where $\mu_{ng}^{out} \in \Omega_{ng}^{out}$ and $\mu_{ng}^{in} \in \Omega_{ng}^{in}$, such that μ_{ng}^{out} and μ_{ng}^{in} are *compatible*. Again, consider μ_{ng}^{out} and μ_{ng}^{in} the pattern solutions where *Val* is deduced from.

Let C be the expression context where dynEnv is included and μ_C the XSPARQL instance mapping of C. Furthermore let P^{in} be the graph pattern obtained from replacing the variables in $WhereClause^{in}$ according

to μ_C. Since we know that $vars\big(WhereClause^{in}\big) \subseteq vars\big(P^{in}\big)$, all solutions mappings returned by evaluating $WhereClause^{in}$ under XSPARQL semantics are included in the pattern solution of evaluating P^{in} under SPARQL semantics i.e. $\Omega_{xs}^{in} \preceq \Omega_{ng}^{in}$. We obtain two cases: (i) $\mu_{ng}^{in} \in \Omega_{xs}^{in}$; or (ii) $\mu_{ng}^{in} \notin \Omega_{xs}^{in}$. In (i) we immediately get that dynEnv $\vdash Q \Rightarrow Val$. For (ii), consider $\mu_{C_1}^{xs}$ the XSPARQL instance of the inner *SparqlForClause* (created based on dynEnv_1^{xs}). As we can see from (T7), dynEnv_1^{xs} (and thus also $\mu_{C_1}^{xs}$) includes the bindings for variables from each solution mapping $\mu_i \in \Omega_{xs}^{out}$. Thus, according to the XSPARQL BGP matching (cf. Definition 4.11), Ω_{xs}^{in} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega_{xs}^{out}$ and specifically those compatible with μ_{ng}^{out}. Since we know that μ_{ng}^{in} is compatible with μ_{ng}^{out}, we have that μ_{ng}^{in} must belong to Ω_{xs}^{in}, thus we can deduce that dynEnv $\vdash Q \Rightarrow Val$.

Inductive Step. Let us assume that, for some arbitrary dynEnv_i, $\text{dynEnv}_i \vdash ExprSingle_1 \Rightarrow Val_i$ if and only if $\text{dynEnv}_i \vdash opt_{ng}(ExprSingle_1) \Rightarrow Val_i$. According to the opt_{ng} rewriting, there must exist a dynEnv_j that is the extension of dynEnv_i with Val_i and thus $\text{dynEnv}_j \vdash ExprSingle_2 \Rightarrow Val$ if and only if $\text{dynEnv}_j \vdash opt_{ng}(ExprSingle_2) \Rightarrow Val$. Consequently, we have that dynEnv $\vdash Q \Rightarrow Val$ if and only if dynEnv $\vdash opt_{ng}(Q) \Rightarrow Val$. □

5.3.3. Nested Queries in XMarkRDF

From the initial set of 20 queries there are 5 queries (q_8–q_{12}) that contain nested expressions. They are described informally in the XMark suite as follows:

(q_8) "List the names of persons and the number of items they bought;"
(q_9) "List the names of persons and the names of the items they bought in Europe;"
(q_{10}) "List all persons according to their interest;"
(q_{11}) "List the number of items currently on sale whose price does not exceed 0.02% of the seller's income;" and
(q_{12}) "For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income."

Figures 5.3a to 5.3c present XMark query q_9, its translated XSPARQL version in XMarkRDB and XMarkRDF, respectively. Query q_9, as presented in Figure 5.3d, is ready to be evaluated by the SPARQL2XQuery system over the XMarkRDF_{S2XQ} dataset.[14]

The different rewritings presented in Section 5.3 can be applied to the four nested queries q_8–q_{11}. Query q_{12} also consists of a nested expression, however the most accurate translation of this query into XSPARQL results in the dependent variable not being *strictly bound* since it occurs only in the `filter` of the inner query. As such, we cannot apply the different rewritings to this query.

XMarkRDF query q_9 is presented in Figure 5.3c. This query is close to queries q_8, q_{10}, and q_{11} and consists of a nested expression: the inner `for` expression of the query (lines 9–13) is executed once for each person matched by the outer expression (lines 6–7), which means that one SPARQL call will be made for each person separately. Thus, the number of SPARQL calls performed in the inner expression directly depends on the size of the dataset (cf. Table 5.1 for details). Queries q_8, q_9, and q_{11} evaluates the inner expression for each person, while q_{10} evaluates the inner expression for each category. Each dataset contains approximately 25 times more persons than categories. The rewriting strategies presented in Section 5.3 reduce the number of SPARQL calls to two: one to get all the people (similar to the direct rewriting version), and one additional SPARQL call for retrieving all the information about all the

[14] Please note that this query follows the syntax presented by Groppe et al. (2008) however, we only had access to the implementation of the translation from SPARQL to XQuery and hence manually replicated the complete query translation.

```
declare ordering unordered;
declare variable $xml external;

let $auction := doc($xml) return
let $ca := $auction/site/closed_auctions/closed_auction
return let $ei := $auction/site/regions/europe/item
for $p in $auction/site/people/person
let $a := for $t in $ca
          where $p/@id = $t/buyer/@person return
          let $n := for $t2 in $ei
                    where $t/itemref/@item = $t2/@id
                    return $t2
return <item>{$n/name/text()}</item>
return <person name="{$p/name/text()}">{$a}</person>
```

(a) Query q_9 in XQuery (XMark)

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare ordering unordered;
declare variable $rdf external;

for person.personid as $person, person.name as $name
from person
return <person name="{$name}">{
    for closed_auctions.closed_auction_id as $ca,
          closed_auctions.itemref as $itemRef
    from closed_auctions
    where closed_auctions.buyer = $person
    return <item>{
      for item.name as $itemname
      from item, region
      where item.region = region.regionid and region.name
            = 'europe' and
          item.itemid = $itemRef
          return $itemname
    }</item>
}</person>
```

(b) Query q_9 in XSPARQL (XMarkRDB)

```
prefix : <http://xsparql.deri.org/data/>
prefix foaf: <http://xmlns.com/foaf/0.1/>
declare ordering unordered;
declare variable $rdf external;

for $person $name from $rdf
where { $person foaf:name $name }
return <person name="{$name}">{
  for * from $rdf where { $ca :buyer $person .
    optional { $ca :itemRef $itemRef .
      $itemRef :locatedIn [ :name "europe" ].
      $itemRef :name $itemname } }
  return <item>{$itemname}</item>
}</person>
```

(c) Query q_9 in XSPARQL (XMarkRDF)

```
declare namespace ac="http://xsparql.deri.org/data/";
declare namespace foaf="http://xmlns.com/foaf/0.1/";
declare variable $rdf external;

for ($n, $m) in
  SELECT $person $name FROM $rdf
  WHERE { $person foaf:name $name . }
return
  <person name="{$n}">{ for ($item) in
     SELECT $itemname WHERE { $ca ac:buyer $person .
       optional { $ca ac:itemRef $itemRef .
         $itemRef ac:locatedIn [ ac:name "europe" ] .
         $itemRef ac:name $itemname } .
    } return <item>{$itemname}</item>
  }</person>
```

(d) Query q_9 in SPARQL2XQuery (XMarkRDF$_{S2XQ}$)

Figure 5.3.: Variants of benchmark query q_9

```
<person name="Alagu Nyrup">
 <item>monument </item>
 <item>herring hush </item>
</person>
```

(a) Query q_9 – bought items grouped by person

```
<item name="monument ">Alagu Nyrup</item>
<item name="herring hush ">Alagu Nyrup</
    item>
```

(b) Query q'_9 – flat list of items and buyer

Figure 5.4.: Example output excerpts of queries q_9 and q'_9

auctions in the dataset. Although the query remains exponential, the practical evaluation will show that reducing the number of SPARQL calls drastically improves query execution times.

As mentioned in Section 5.3.2, for the SPARQL based rewritings, we want the query output to be computable directly in SPARQL without any further processing, i.e. we do not want to use XQuery for further processing of the SPARQL results and the query should be expressible in SPARQL without features from SPARQL 1.1. Since the original nested queries q_8–q_{11} group the output results (while optionally applying some aggregation function), we need to include modified versions of these benchmark queries for the evaluation of the SPARQL based rewritings. In these modified queries, denoted q'_8–q'_{11}, we changed the return format of the queries to consist of a flattened representation of the output of the original query. An example of the output for queries q_9 and q'_9 is presented in Figure 5.4. All queries q'_i and q''_i follow a similar strategy for reformatting the output: the queries resulting from applying opt_{sr} are named q'_8–q'_{11}, while the queries that consist of an outer `for` expression – to which opt_{ng} was applied – are q''_8–q''_{11}.

Table 5.5.: Query response times (in seconds) of different optimisations for the 2MB datasets. Optimisation not applicable (n/a).

	XS^{rdf}	XS^{rdb}	$S2XQ$	XS^{rdf}_{nl}	XS^{rdb}_{nl}	$S2XQ_{nl}$	XS^{rdf}_{sr}	$S2XQ_{sr}$	XS^{rdf}_{ng}
q_8	293.62	**1.27**	1.28	5.37	1.74	1.82	n/a	n/a	n/a
q_9	292.84	**1.56**	11.57	5.37	2.81	12.35	n/a	n/a	n/a
q_{10}	16.92	**1.62**	309.21	5.94	2.30	88.73	n/a	n/a	n/a
q_{11}	295.19	**2.33**	102.42	11.43	2.56	97.01	n/a	n/a	n/a
q'_8	291.99	n/a	1.23	6.03	n/a	1.79	3.43	**1.23**	n/a
q'_9	292.49	n/a	11.49	5.38	n/a	12.34	**3.56**	8.59	n/a
q'_{10}	16.86	n/a	307.42	5.94	n/a	87.68	**5.07**	—	n/a
q'_{11}	295.68	n/a	101.54	12.27	n/a	96.54	**5.88**	120.42	n/a
q''_8	293.64	n/a	60.77	5.11	n/a	**1.54**	n/a	n/a	4.76
q''_9	292.58	n/a	—	**4.91**	n/a	9.83	n/a	n/a	4.99
q''_{10}	6.78	n/a	417.97	**5.19**	n/a	115.95	n/a	n/a	5.27
q''_{11}	296.12	n/a	91.03	12.29	n/a	95.17	n/a	n/a	**8.59**

5.3.4. Evaluation of the Proposed Optimisations

In this section we present an experimental evaluation of the different rewritings presented in Section 5.3. For this evaluation we also rely on the XMarkRDF benchmark suite (presented in Section 5.2) and compare, when possible, the effects of the different rewritings on the SPARQL2XQuery system (Groppe et al., 2008).

For the evaluation we extend the run configurations presented in Section 5.2 with the following:

XS^{rdf}_Z: using the XSPARQL implementation over the XMarkRDF datasets (translated data and queries) with nested expresion optimisation opt_Z for $Z \in \{ nl, ng, sr \}$;

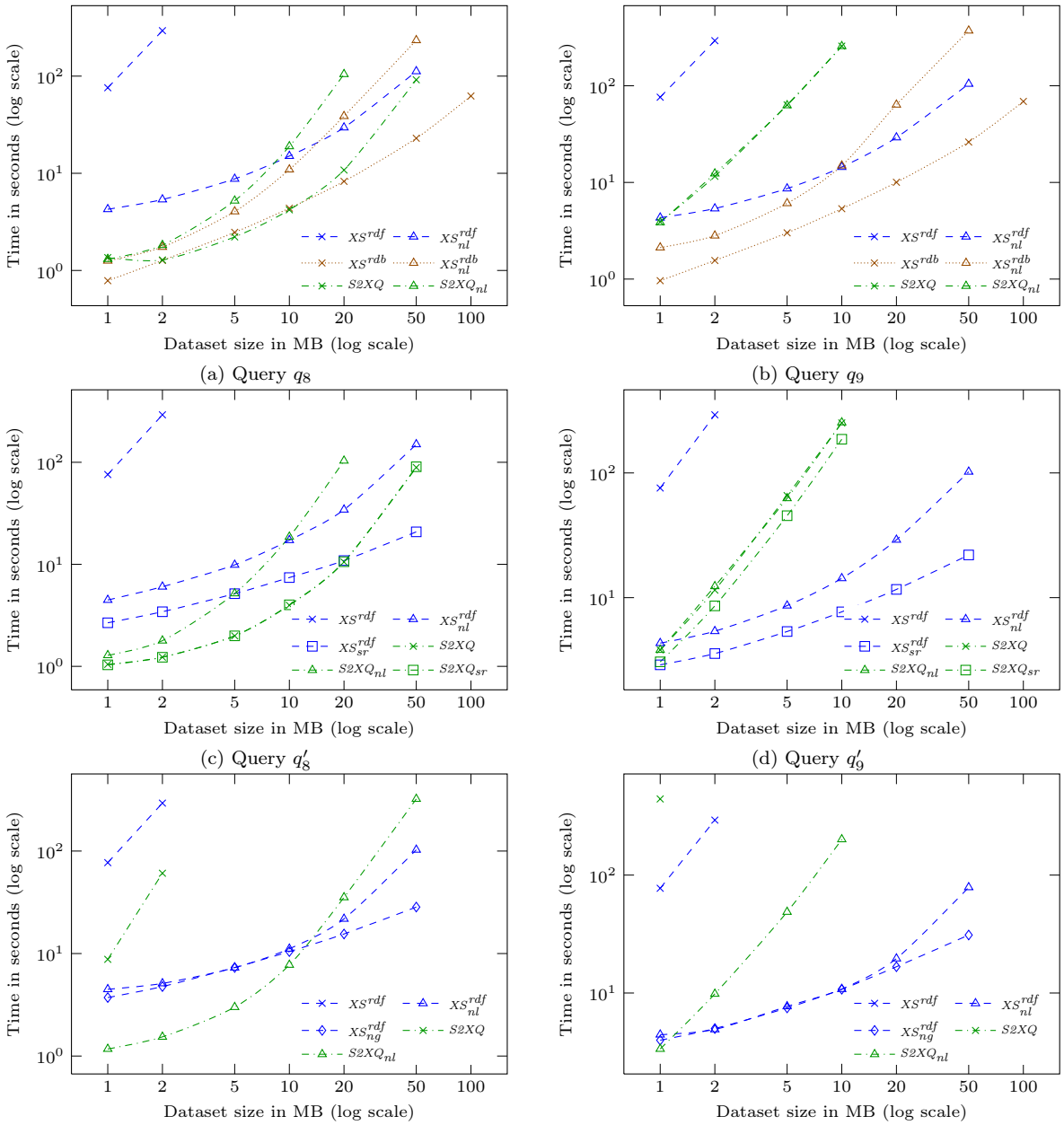
XS^{rdb}_{nl}: using the XSPARQL implementation over the XMarkRDB datasets (translated data and queries) with nested expresion optimisation opt_{nl};

$S2XQ_Z$: using the SPARQL2XQuery implementation over the translation of the XMarkRDF datasets into the required XML format (XMarkRDF_{S2XQ}) with nested expresion optimisation opt_Z for $Z \in \{ nl, sr \}$.

The experimental setup remains the same as presented in Section 5.2.1. We applied the nested-loop join rewriting from Section 5.3.1 to the XMarkRDB and XMarkRDF translated queries, which are denoted as XS^{rdb}_{nl} and XS^{rdf}_{nl}, respectively. The same optimisations were applied to the SPARQL2XQuery translation to XQuery, denoted $S2XQ_{nl}$ in the results. The strategies of rewriting to a single SPARQL query, as presented in Section 5.3.2, were also applied to the XSPARQL XMarkRDF and SPARQL2XQuery queries and are denoted as XS^{rdf}_{sr} and $S2XQ_{sr}$, respectively. The Named Graph rewriting was applied to the XSPARQL XMarkRDF queries and is denoted XS^{rdf}_{ng}.

The comparison of the response times of the different rewriting functions presented in Section 5.3 is shown graphically in Figures 5.5 and 5.6. The response times of these queries for the 2MB are presented in Table 5.5 as a reference, where n/a indicates that the combination of query and optimisation is not applicable.

As we can see from Table 5.5 and Figures 5.5 and 5.6, the opt_{nl} optimisation provides significant reduction in the query evaluation times when applied to the nested queries with an inner *SparqlForClause*. For queries q_8, q_9, and q_{11} the difference in response times is one order of magnitude. However, applying a similar rewriting to relational data deteriorates the response times of the query. This hints that collecting



(a) Query q_8 (b) Query q_9

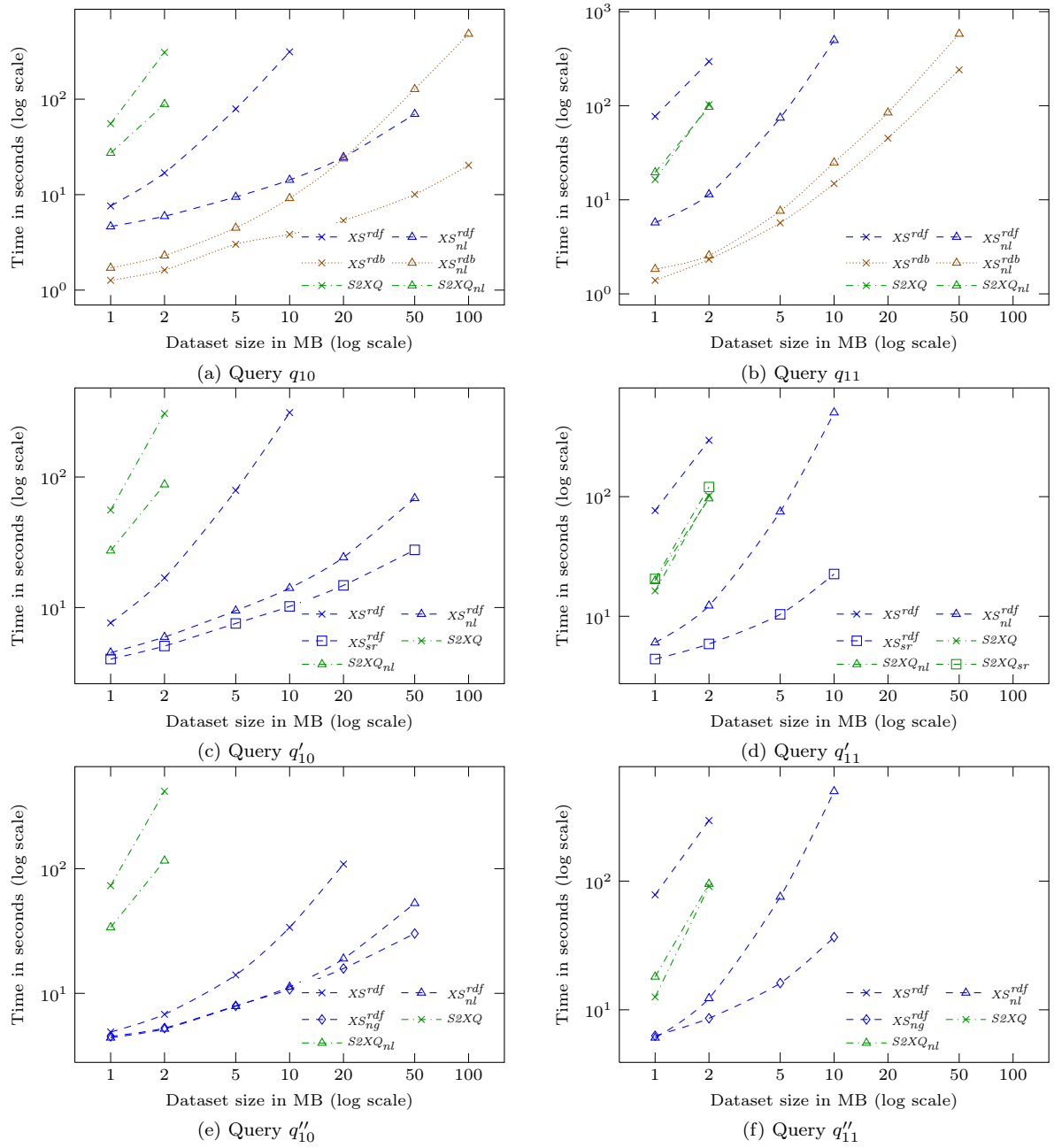
(c) Query q'_8 (d) Query q'_9

(e) Query q''_8 (f) Query q''_9

Figure 5.5.: Query response times for (variants of) q_8 and q_9 on all XMarkRDF datasets

the data and performing the join in the rewritten XQuery is slower than the nested calls to the relational database. There are two possible causes for this discrepancy of behaviours from the different backends. One possible explanation for the speed improvement in SPARQL is that the overhead resides on the loading of data by the ARQ engine. Since this overhead is not presented in the relational database the queries would not benefit from the optimised rewritings. The other explanation is that the cost of evaluating one unbound query and building the necessary structures for representing the returned data is greater than the cost of executing nested queries. Further investigation into this would be required to determine the source of the overhead.

The improvement in the execution time for query q_{10} is less drastic. This can be explained by the fact that the outer expression of q_{10} iterates over "categories", which, as presented in Table 5.1, increases at a



(a) Query q_{10} (b) Query q_{11}

(c) Query q'_{10} (d) Query q'_{11}

(e) Query q''_{10} (f) Query q''_{11}

Figure 5.6.: Query response times for (variants of) q_{10} and q_{11} on all XMarkRDF datasets

much smaller rate than "persons" do in the outer expressions of queries q_8, q_9, and q_{11}.

However, for the *S2XQ* runs this optimisation provides virtually no improvement in the query response times for queries q_8 and q_9 and their variants. In queries q_{10}, q_{11}, q'_{10}, and q'_{11} we can observe an improvement in response times. This can be attributed to the fact that the rewriting for queries q_{10} and q_{11} and their variants are not as suitable for optimisation by the XQuery engine when compared to queries q_8 and q_9. For these cases our rewriting strategy is capable of performing the optimisation task for the XQuery engine.

For the XS^{rdf} run, it is possible to see in Figures 5.5c and 5.5d that opt_{sr} (presented in Section 5.3.2) is generally more efficient in terms of response times than the XQuery based. This can be justified by the the smaller amount of information that is necessary to transfer from SPARQL to the XQuery engine.

```
prefix : <http://example.org/bands#>
prefix foaf: <http://xmlns.com/foaf/0.1/>

construct { _:{fn:replace($band, "http://dbpedia.org/resource/","")} foaf:name $name ;
     foaf:member $member }
from <file:bands.ttl>
where { $band a mo:MusicGroup; foaf:name $name; foaf:member $member }
```

Query 5.3: Transformation between RDF representations in XSPARQL

```
prefix : <http://example.org/bands#>
prefix foaf: <http://xmlns.com/foaf/0.1/>

CONSTRUCT { $b foaf:name $name ; foaf:member $member }
FROM <file:usecaseData.ttl>
WHERE
  {
   { SELECT DISTINCT $band $name (BNODE() as $b)
     WHERE { $band foaf:name $name }
   }
   $band foaf:member $member
   }
```

Query 5.4: Transformation between RDF representations in SPARQL 1.1

This effectively reduces the overhead of using an external SPARQL engine for the evaluation of queries. Considering the *S2XQ*$_{sr}$ run, opt_{sr} produces no improvement in the query response times and in some cases (q'_{10} and q'_{11} from Table 5.5) even deteriorates considerably the response times when compared to *S2XQ*. This further supports our previous claims that the XQuery engine is not capable of optimising the rewritten code from complex SPARQL queries.

Furthermore, the *S2XQ*$_{sr}$ runs could only evaluate the smaller dataset sizes for query q_8: its response times deteriorated considerably with the larger dataset sizes, as opposed to the XS^{rdf}_{sr} runs that behaved consistently similar to XS^{rdf}_{nl}. This indicates that *S2XQ* is not as efficient as the ARQ-based native SPARQL engine runs XS^{rdf}_{sr} and XS^{rdf}_{nl} for larger datasets.

We can draw similar conclusions for the opt_{ng} when comparing the query evaluation times of the opt_{sr} rewriting. However, the response times for this approach are deteriorated by the the overhead of creating, inserting and deleting the RDF Named Graph. This slowdown makes queries q''_8, q''_{10} and q''_{11} of the of the opt_{nl} rewriting outperform this optimisation.

5.4. Related Work

In this related work section we present a comparison of different approaches for nesting in SPARQL, including the proposal from the new version of SPARQL, SPARQL 1.1.

The new version of SPARQL was presented in Section 3.3 and introduces many new features that we already support in XSPARQL. Most notably: (i) construction of values in `select` expressions; (ii) variable assignment; (iii) remote endpoint querying; and (iv) subqueries (or query nesting). The value construction and variable assignment features behave similar to the features in XSPARQL. However, the XSPARQL language provides a convenient syntax for mapping between different RDF vocabularies, e.g. by generating blank node labels, a task that can otherwise be cumbersome even in SPARQL 1.1.

Example 5.3 (Translation between RDF vocabularies)**.** Query 5.3 presents a simple transformation between different RDF vocabularies: from using URIs to identify bands to blank nodes, where we create a blank node that identifies each band and replicate its members. A more involved but equivalent query written in SPARQL 1.1 is presented in Query 5.4.

Subqueries (along with the special form of subquery that is the remote endpoint query) still presents noteworthy differences between our approach in XSPARQL and the approach proposed in SPARQL 1.1. These differences, also briefly highlighted in Section 5.1.1, are mostly related to the evaluation method of the different languages, while SPARQL follows a bottom up approach, XQuery and thus XSPARQL follow a top-down approach. In SPARQL, the subqueries are evaluated and the produced bindings are then merged with the bindings from the outer query and such subqueries must be executed over the same dataset as the outer query, i.e. `from` and `from named` clauses are not allowed in subqueries. However, this evaluation method prevents the reuse of variables declared in the inner query. The different evaluation models cater for different and complementary use cases. While the method followed by SQL and SPARQL is suitable for parallel and distributed computing, the model followed by XQuery (and thus XSPARQL), allowing to inject values into the inner expressions, can be a necessary feature.

Also related to our nested queries optimisation, Angles and Gutiérrez (2010) presented initial work on an extension of SPARQL that caters for nested queries and presented preliminary equivalences between types of nested queries with the aim of determining if query unnesting can be successfully applicable. The same authors then extended this work in Angles and Gutiérrez (2011), where they consider different forms of nesting, namely nesting in `from` clauses, nesting in graph patterns, and nesting in filter expressions. In XSPARQL, we easily support the nesting in `from` clauses by assigning the result of a `construct` query to a variable and reusing this variable in a `from` clause. Nesting in graph patterns can also be simulated in XSPARQL by using the standard XQuery nesting of expressions, in this case nesting *SparqlForClause*s. Regarding nesting in `filter` expressions, although possible to implement in XSPARQL, this would require processing of the results from the SPARQL query in XSPARQL.

The presented approaches for query rewriting applied to XSPARQL nested queries is similar to already known optimisations from the relational databases realm and also presented for XQuery queries by May et al. (2003). This work proposes unnesting equivalences that, while maintaining the element order, provide significant performance gains. For the original XMark nested queries q_8 and q_{11}, we can consider using the equivalences described for the "Grouping and Aggregation" unnesting equivalences, whereas q_9 and q_{10} rely on the "Grouping" equivalences. Hence, the nested-loop rewriting of the queries we present replicates the unnested query plans for these optimisations.

5.5. Conclusion

In this chapter we presented our implementation of the XSPARQL language presented in Chapter 4. Our implementation attempts to reuse *off-the-shelf* components, where we translate each XSPARQL query into an XQuery containing interleaved calls to a SQL and/or SPARQL engine. The benchmark evaluation of our implementation has shown that nested queries incur the highest evaluation overhead and thus we presented different rewritings that aim at reducing this overhead.

The presented optimisations are based on reordering the expressions in the XQuery rewriting to minimise the number of calls to the SPARQL endpoint or based on performing a more complex SPARQL query that takes care of joining the variables. The benchmarks that were carried out to determine the impact of our optimisations have shown encouraging results for nested expressions whose inner expressions access RDF data, hinting at a large potential for optimisations in XSPARQL. However, similar rewritings do not produce the excepted improvements for nested expressions that access relational data. This

indicates that different optimisations need to be considered for accessing relational data.

Among the rewriting strategies presented in this chapter and on our test data, pushing joins into a SPARQL engine appeared the most promising strategy. Our benchmark results showed that our optimisations are not only specific to XSPARQL having also improved the response times of the SPARQL2XQuery system to which we compared XSPARQL.

Also according to our benchmarks, encoding SPARQL in XQuery seems a viable option – assuming that we have access to the RDF dataset beforehand – that would allow to compile XSPARQL to pure XQuery without the use of a separate SPARQL engine.

6. An Extension of RDF and SPARQL towards Meta-Information

In this chapter, we present an extension of the RDF model to support meta-information in the form of annotations attached to RDF triples. On a high-level, we attach this meta-information to an RDF triple according to a predetermined annotation domain: *temporal*, *fuzzy*, *provenance*, and possibly others (as we will see in the next chapter). We specify the semantics of this extension by conservatively extending the RDFS semantics and provide a deductive system for Annotated RDFS. Furthermore, we define a query language that extends SPARQL to query this meta-information and include advanced features such as aggregates, nested queries and variable assignments, which are part of the SPARQL 1.1 specification.

Meta-information about relational tuples was investigated as an important aspect of the relational model. For instance, maintaing temporal information for representing the validity of the triple or provenance information to determine the origin of tuples. Similarly, meta-information in RDF is similar importance. Temporal information was acknowledged in the W3C RDF specification (P. Hayes, 2004) but deliberately left out, stating:

> "There are several aspects of meaning in RDF which are ignored by this semantics; in particular, it treats URI references as simple names, ignoring aspects of meaning encoded in particular URI forms and does not provide any analysis of time-varying data or of changes to URI references."

The W3C provenance working group is also investigating how to define a vocabulary that allows for provenance information to be interchanged and also to attach provenance information to specific RDF resources (Belhajjame et al., 2012).

For the context of this thesis, we are particularly interested in the role that meta-information can play in a data integration system, for example by allowing to resolve conflicts in the integrated information. This is acknowledged by Halevy, Rajaraman et al. (2006), who identify the inclusion of uncertain and provenance information during data integration as a challenge that needs to be overcome.

Meta-Information in Legacy Data Models

Shortly after the introduction of the relational model research began to focus on extending it towards temporal information (Wiederhold et al., 1975; Snodgrass, 1990; Abiteboul, Hull et al., 1995; Snodgrass, 1999). Temporal information is commonly attached to relational tuples in order to represent *valid time*: the time period for which the information that a tuple represents is considered valid. Another form of temporal information that can be attached to tuples is the so-called *transaction time*, where information regarding when a specific tuple was inserted into the relational database is stored. This form of temporal information is important specifically for defining operations like *transactions* and *rollbacks* of information in a database management system, e.g. reverting the contents of the database to a specific time. In the present chapter we are concerned only with validity time. Regarding query languages, a temporal extension of the SQL query language, named TSQL2, was also presented by Snodgrass et al. (1994). This extension aims at being compatible with the SQL query language and introduces datatypes and keywords to the language that allow to query the temporal aspects of the database.

Similar extensions have also been proposed for XML (Amagasa et al., 2000; Rizzolo and Vaisman, 2008). Amagasa et al. (2000) propose an extension of the XDM where edges are labelled with a time validity and consider a hierarchical time structure: the time validity of all the children of a node must be contained within the time validity of the current node and among siblings there cannot exist time intersection. Targeted at modelling transaction time, a similar approach is followed by Rizzolo and Vaisman (2008). Furthermore, the authors present TXPath, an extension to the XPath query language to support the new temporal XML data.

However temporal information is not the only kind of meta-information we can represent and other extensions to the relational model also allow to represent ambiguous or approximate data in the form of fuzzy information. An overview of fuzzy databases is provided by Ma and Yan (2008), and also for fuzzy meta-information, extensions were proposed for XML (Ma and Yan, 2007).

Other extensions to querying XML, also related to the XSPARQL language, include the SXPath language (Oro et al., 2010), which focuses on information extraction from HTML pages and thus provides spatial extensions of the XPath language that allow to locate elements in the rendered HTML page.

Meta-Information in RDF

Several extensions of RDF were proposed in order to deal with time (Gutiérrez, Hurtado and Vaisman, 2007; Pugliese et al., 2008; Tappolet and A. Bernstein, 2009), truth or imprecise information (Mazzieri and Dragoni, 2008; Straccia, 2009), trust (Hartig, 2009; Schenk, 2008), and combinations of the previous extensions (Dividino et al., 2009). All of these proposals share a common approach: extending the RDF language by attaching meta-information about the RDF graph or about individual triples.

The basis of Annotated RDFS, allowing to represent the kinds of meta-information we have described, were first established by Udrea et al. (2006); Udrea et al. (2010), where the authors introduce RDF triples annotated with values taken from an *annotation domain*, defined as a *finite partial order*. This annotation domain may contain information regarding the temporal validity of the triple or the level of uncertainty of the triple. Notably, the inference capabilities presented in their work are limited to a small subset of RDFS.[1]

In this chapter we introduce a richer, not necessarily finite, structure that is backwards-compatible with RDF and RDFS. Our proposed inference system, in the form of an extension of the RDFS rules, provides support for more inference rules when compared to Udrea et al. (2010) and also a more fine-grained propagation of annotation values through the inferred triples. Furthermore we introduce an extension of SPARQL (Prud'hommeaux and Seaborne, 2008), the W3C-standardised query language for RDF (cf. Section 3.3), that allows us to query this extended representation of RDF triples. Although the respective RDF graphs, datasets, and queries are domain-specific, i.e. the annotations included in these graphs and queries must correspond to a specific domain, the proposed extension of the RDFS rules and SPARQL query language is domain-independent, i.e. we can define this as an extension that covers all domains.

6.1. RDF(S) with Annotations

For extending our running example we use the temporal domain, which allows us to annotate the RDF data with temporal information. For instance, we can annotate the band members' triples to reflect their active years with the band. A possible temporal query that can be easily performed over data represented in this format is "What were the members of a band at a certain time?" The use of other

[1] To distinguish our work from the original Annotated RDF by Udrea et al., we call our framework Annotated RDFS. However, when referring to specific graphs we will keep the original Annotated RDF name.

```
@prefix ex: <http://example.org/bands#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dbpedia: <http://dbpedia.org/resource/> .
@prefix mo: <http://purl.org/ontology/mo/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

dbpedia:Nightwish a mo:MusicGroup "[1996,2012]" .
dbpedia:Nightwish foaf:name "Nightwish" .
dbpedia:Nightwish foaf:member dbpedia:Marco_Hietala "[2001,2012]" .
dbpedia:Nightwish foaf:member dbpedia:Tarja_Turunen "[1996,2005]" .
dbpedia:Marco_Hietala foaf:name "Marco Hietala"@en "[1966,2012]" .
dbpedia:Marco_Hietala a mo:MusicArtist "[1984,2012]" .
dbpedia:Tarja_Turunen foaf:name "Tarja Turunen"@en "[1977,2012]" .
dbpedia:Tarja_Turunen a mo:MusicArtist .
ex:album208 a mo:Record "[2000,2012]" .
ex:album208 mo:title "Wishmaster" .
ex:album208 foaf:maker dbpedia:Nightwish .
ex:album208 mo:track _:song566 .
ex:album208 mo:track _:song506 .
_:song566 a mo:Track .
_:song566 dc:title "Wishmaster" .
_:song506 a mo:Track .
_:song506 dc:title "FantasMic" .
```

Data 6.1: Temporal Annotated RDFS

domains would allow to represent different views on the data, for example in the fuzzy domain, we can represent information regarding part-time members of bands.

Data 6.1 represents an extension of our use case data from Data 2.5 annotated with information from the temporal domain, which intuitively means that the annotated triple is valid in dates contained in the annotation interval (the exact meaning of the annotations will be explained later). In Data 6.1, we are representing the annotated triples using N-Quads (Cyganiak et al., 2009), a format that allows to attach a forth element to an RDF triple.[2] However, in examples and definitions of the rest of this chapter we will use a representation of the form $(s, p, o) : \lambda$, which is considered equivalent to its N-Quad counterpart.

6.1.1. Syntax

Our approach is to extend triples with annotations, where an annotation is taken from a specific domain. This extension follows a similar approach to the annotated logic programming framework (Kifer and Subrahmanian, 1992).

Definition 6.1 (Annotated RDF triple and graph)**.** *An* annotated triple *is an expression* $\tau : \lambda$*, where* τ *is an RDF triple and* λ *is an* annotation value *(defined below). An* annotated graph *is a finite set of annotated triples. Furthermore we call an annotated graph* G *a* normalised annotated graph *iff* $\nexists \tau : \lambda_1, \tau : \lambda_2 \in G$ *s.t.* $\lambda_1 \neq \lambda_2$.

The intended semantics of annotated triples depends of course on the meaning we associate to the annotation values. For instance, in a temporal setting (Gutiérrez, Hurtado and Vaisman, 2007),

$$(\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, \texttt{dbpedia:Marco\_Hietala}) : [2001, 2012]$$

has the intended meaning "Marco Hietala was a member of Nightwish during the period from 2001 to 2012", while in the fuzzy setting (Straccia, 2009), we can represent part-time members of a band:

$$(\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, \texttt{dbpedia:Troy\_Donockley}) : 0.7$$

[2] A similar approach is followed for extending SPARQL's syntax.

with the intended meaning "Troy Donockley is a member of Nightwish to a degree not less than 0.7".[3]

6.1.2. Annotation Domain Specification

To start with, let us consider a non-empty set L, where the elements in L are our annotation values. For example, in a fuzzy setting, $L = [0, 1]$, while in a typical temporal setting, L may be time points or time intervals. In our annotation framework, we extend the notion of interpretation (presented in Definition 2.6) to map statements to elements of the annotation domain. But first let us define an annotation domain:

Definition 6.2 (Annotation Domain)**.** *We say that an* annotation domain *for RDFS is an idempotent, commutative semi-ring*

$$D = \langle L, \oplus, \otimes, \bot, \top \rangle \ ,$$

where, $\top, \bot$ are specific annotation values and $\oplus$ is $\top$-annihilating (Buneman and Kostylev, 2010). That is, for $\lambda, \lambda_i \in L$:

1. *$\oplus$ is idempotent, commutative, associative;*
2. *$\otimes$ is commutative and associative;*
3. *$\bot \oplus \lambda = \lambda$, $\top \otimes \lambda = \lambda$, $\bot \otimes \lambda = \bot$, and $\top \oplus \lambda = \top$;*
4. *$\otimes$ is distributive over $\oplus$, i.e. $\lambda_1 \otimes (\lambda_2 \oplus \lambda_3) = (\lambda_1 \otimes \lambda_2) \oplus (\lambda_1 \otimes \lambda_3)$.*

Please note that there is a natural partial order on any idempotent semi-ring: an annotation domain $D = \langle L, \oplus, \otimes, \bot, \top \rangle$ induces a partial order $\preceq$ over L defined as:

$$\lambda_1 \preceq \lambda_2 \ \text{ if and only if } \lambda_1 \oplus \lambda_2 = \lambda_2 \ .$$

The $\top$ and $\bot$ respective represent the highest and lowest element in the partial order. This partial order $\preceq$ is used to express redundant information: for instance, for temporal intervals, an annotated triple $(s, p, o) : [2000, 2006]$ includes $(s, p, o) : [2003, 2004]$, since $[2003, 2004] \subseteq [2000, 2006]$ (here, $\subseteq$ plays the role of $\preceq$).

In previous work (Straccia et al., 2010; Lopes, Polleres et al., 2010), an annotation domain was assumed to be a more specific structure, namely a residuated bounded lattice. In Buneman and Kostylev (2010) it was shown that we may use a slightly weaker structure than residuated lattices for annotation domains.

We use $\oplus$ to combine information about the same statement. For instance, in temporal logic, from $\tau : [2000, 2006]$ and $\tau : [2003, 2008]$, we infer $\tau : [2000, 2008]$, as $[2000, 2008] = [2000, 2006] \cup [2003, 2008]$ (where $\cup$ plays the role of $\oplus$). In the fuzzy context, from $\tau : 0.7$ and $\tau : 0.6$, we infer $\tau : 0.7$, since $0.7 = \max(0.7, 0.6)$ (here, max plays the role of $\oplus$).

We use $\otimes$ to model the "conjunction" of information, where $\otimes$ is a generalisation of boolean conjunction to the many-valued case. In fact, $\otimes$ satisfies also that:

1. $\otimes$ is bounded: i.e. $\lambda_1 \otimes \lambda_2 \preceq \lambda_1$.
2. $\otimes$ is $\preceq$-monotone, i.e. for $\lambda_1 \preceq \lambda_2$, $\lambda \otimes \lambda_1 \preceq \lambda \otimes \lambda_2$

For instance, on interval-valued temporal logic, from $(a, \mathsf{sc}, b) : [2000, 2006]$ and $(b, \mathsf{sc}, c) : [2003, 2008]$, we will infer $(a, \mathsf{sc}, c) : [2003, 2006]$, as $[2003, 2006] = [2000, 2006] \cap [2003, 2008]$ (where $\cap$ plays the role of $\otimes$). In the fuzzy context, one may chose any triangular norm (t-norm) (Klement et al., 2000), e.g. product, and, thus, from $(a, \mathsf{sc}, b) : 0.7$ and $(b, \mathsf{sc}, c) : 0.6$, we will infer $(a, \mathsf{sc}, c) : 0.42$, as $0.42 = 0.7 \cdot 0.6$ (here, $\cdot$ plays the role of $\otimes$).

[3] The membership degree was chosen as an example, Troy has collaborated with Nightwish on different albums and live concerts.

The distributivity condition guarantees that we obtain the same annotation $\lambda_1 \otimes (\lambda_2 \oplus \lambda_3) = (\lambda_1 \otimes \lambda_2) \oplus (\lambda_1 \otimes \lambda_3)$ of the triple (a, sc, c) that can be inferred from triples $(a, \mathsf{sc}, b) \colon \lambda_1$, $(b, \mathsf{sc}, c) \colon \lambda_2$ and $(b, \mathsf{sc}, c) \colon \lambda_3$. Finally, note that, conceptually, in order to build an annotation domain, one has to:

1. determine the set of annotation values L (typically a countable set[4]), identifying the top and bottom elements;
2. define suitable operations $\otimes$ and $\oplus$ that acts as "conjunction" and "disjunction" functions, to support the intended inference over schema axioms, such as

   "from $(a, \mathsf{sc}, b) \colon \lambda$ and $(b, \mathsf{sc}, c) \colon \lambda'$ infer $(a, \mathsf{sc}, c) \colon \lambda \otimes \lambda'$"

   and

   "from $\tau \colon \lambda$ and $\tau \colon \lambda'$ infer $\tau \colon \lambda \oplus \lambda'$"

Another desirable feature is to use annotated and non-annotated triples in parallel, possibly even in the same dataset. In Zimmermann et al. (2012), we presented several approaches for combining annotated and non-annotated triples, such as assuming a default annotation for any non-annotated triple or creating a new compound domain. For simplicity, and since we are considering the issue of compound domains as out of scope for this thesis, we follow the approach of assuming a default annotation for non-annotated triples. This default annotation can be specified on a per-domain basis however, if unspecified, we assume the $\top$ element from the domain as the default annotation.

6.1.3. Semantics

For this section, we fix an annotation domain $D = \langle L, \oplus, \otimes, \bot, \top \rangle$. Similar to Section 2.4.2 we rely on the ρdf fragment of RDFS and do not consider datatype interpretations.

Definition 6.3 (Annotated Map)**.** *An* annotated map *is a function $\theta : \mathbf{UBL} \to \mathbf{UBL}$ preserving URIs and literals, i.e. $\theta(t) = t$, for all $t \in \mathbf{UL}$. Given an annotated graph G, we define $\theta(G) = \{ (\theta(s), \theta(p), \theta(o)) \colon \lambda' \mid (s, p, o) \colon \lambda \in G \}$, where $\lambda' \in \mathbf{L}$ and $\lambda' \preceq \lambda$. Similarly to the classical case, we speak of an annotated map θ from G_1 to G_2, and write $\theta : G_1 \to G_2$, if θ is such that $\forall \tau \colon \lambda_2 \in G_2, \exists \tau \colon \lambda_1 \in G_1$ such that $\lambda_2 \preceq \lambda_1$.*[5]

Informally, an interpretation $\mathcal{I}$ will assign to a triple τ an element of the annotation domain $\lambda \in L$:

Definition 6.4 (Annotated Interpretation, extends Definition 2.6)**.** *An* annotated interpretation $\mathcal{I}$ *over a vocabulary V is a tuple*

$$\mathcal{I} = \langle \Delta_R, \Delta_P, \Delta_C, \Delta_L, P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}} \rangle$$

where $\Delta_R, \Delta_P, \Delta_C, \Delta_L$ are interpretation domains of $\mathcal{I}$ and $P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}}$ are interpretation functions of $\mathcal{I}$. They have to satisfy:

1. *Δ_R is a nonempty finite set of resources, called the domain or universe of $\mathcal{I}$;*
2. *Δ_P is a finite set of property names (not necessarily disjoint from Δ_R);*
3. *$\Delta_C \subseteq \Delta_R$ is a distinguished subset of Δ_R identifying if a resource denotes a class of resources;*
4. *$\Delta_L \subseteq \Delta_R$, the set of literal values, Δ_L contains all plain literals in $\mathbf{L} \cap V$;*
5. *$P[\![\cdot]\!]$ maps each property name $p \in \Delta_P$ into a function $P[\![p]\!] : \Delta_R \times \Delta_R \to L$, i.e. assigns an annotation value to each pair of resources;*

[4] Note that one may use XML decimals in $[0, 1]$ in place of real numbers for the fuzzy domain.
[5] As a shorthand notation, from herein we will use $G_2 \preceq G_1$ to denote $\forall \tau \colon \lambda_2 \in G_2, \exists \tau \colon \lambda_1 \in G_1$ such that $\lambda_2 \preceq \lambda_1$.

6. $C[\![\cdot]\!]$ *maps each class* $c \in \Delta_C$ *into a function* $C[\![c]\!] : \Delta_R \to L$*, i.e. assigns an annotation value representing class membership in* c *to every resource;*

7. $\cdot^{\mathcal{I}}$ *maps each* $t \in \mathbf{UL} \cap V$ *into a value* $t^{\mathcal{I}} \in \Delta_R \cup \Delta_P$ *and such that* $\cdot^{\mathcal{I}}$ *is the identity for plain literals and assigns an element in* Δ_R *to each element in* $\mathbf{L}$*.*

Similar to the classical case we provide a single notion of interpretation that covers Simple, RDF, and RDFS (ρdf) entailment. Furthermore, we extend the definition of model:

Definition 6.5 (Model, extends Definition 2.7)**.** *An interpretation* $\mathcal{I}$ *is a* model *of an annotated ground graph* G*, denoted* $\mathcal{I} \models G$*, if and only if* $\mathcal{I}$ *is an interpretation over the vocabulary* $\rho df \cup universe(G)$ *that satisfies the following conditions, where* $A\colon \mathbf{B} \mapsto \Delta_R$ *and* $\mathcal{I}_A$ *is the extension of* $\mathcal{I}$ *with* A*:*

Simple:

1. $(s, p, o)\colon \lambda \in G$ *implies* $p^{\mathcal{I}_A} \in \Delta_P$ *and* $P[\![p^{\mathcal{I}_A}]\!](s^{\mathcal{I}_A}, o^{\mathcal{I}_A}) \succeq \lambda$*;*

Subproperty:

1. $P[\![\mathsf{sp}^{\mathcal{I}}]\!](p, q) \otimes P[\![\mathsf{sp}^{\mathcal{I}}]\!](q, r) \preceq P[\![\mathsf{sp}^{\mathcal{I}}]\!](p, r)$*;*
2. $P[\![p^{\mathcal{I}}]\!](x, y) \otimes P[\![\mathsf{sp}^{\mathcal{I}}]\!](p, q) \preceq P[\![q^{\mathcal{I}}]\!](x, y)$*;*

Subclass:

1. $P[\![\mathsf{sc}^{\mathcal{I}}]\!](c, d) \otimes P[\![\mathsf{sc}^{\mathcal{I}}]\!](d, e) \preceq P[\![\mathsf{sc}^{\mathcal{I}}]\!](c, e)$*;*
2. $C[\![c^{\mathcal{I}}]\!](x) \otimes P[\![\mathsf{sc}^{\mathcal{I}}]\!](c, d) \preceq C[\![d^{\mathcal{I}}]\!](x)$*;*

Typing I:

1. $C[\![c]\!](x) = P[\![\mathsf{type}^{\mathcal{I}}]\!](x, c)$*;*
2. $P[\![\mathsf{dom}^{\mathcal{I}}]\!](p, c) \otimes P[\![p]\!](x, y) \preceq C[\![c]\!](x)$*;*
3. $P[\![\mathsf{range}^{\mathcal{I}}]\!](p, c) \otimes P[\![p]\!](x, y) \preceq C[\![c]\!](y)$*;*

Typing II:

1. *For each* $\mathsf{e} \in \rho df$*,* $\mathsf{e}^{\mathcal{I}} \in \Delta_P$*;*
2. $P[\![\mathsf{sp}^{\mathcal{I}}]\!](p, q)$ *is defined only for* $p, q \in \Delta_P$*;*
3. $C[\![\mathsf{sc}^{\mathcal{I}}]\!](c, d)$ *is defined only for* $c, d \in \Delta_C$*;*
4. $P[\![\mathsf{dom}^{\mathcal{I}}]\!](p, c)$ *is defined only for* $p \in \Delta_P$ *and* $c \in \Delta_C$*;*
5. $P[\![\mathsf{range}^{\mathcal{I}}]\!](p, c)$ *is defined only for* $p \in \Delta_P$ *and* $c \in \Delta_C$*;*
6. $P[\![\mathsf{type}^{\mathcal{I}}]\!](s, c)$ *is defined only for* $c \in \Delta_C$*.*

Intuitively, a triple $(s, p, o)\colon \lambda$ is satisfied by $\mathcal{I}$ if (s, o) belongs to the extension of p to a "wider" extent than λ. Note that the major differences from the classical setting reside on items 5 and 6.

Finally, entailment among annotated ground graphs G and H is as usual. Now, $G \models H$, where G and H may contain blank nodes, if and only if for any grounding G' of G there is a grounding H' of H such that $G' \models H'$.

Please note that we always have that $G \models \tau\colon \bot$, however, triples of the form $\tau\colon \bot$ are uninteresting and, thus, in the following we do not consider them as part of the language.

6.1.4. Examples of Annotation Domains

Next we specify some domains in Annotated RDFS, namely the classical domain, fuzzy (Straccia, 2009), temporal (Gutiérrez, Hurtado and Vaisman, 2007), and provenance.

The Classical Domain

The classical RDF setting corresponds to the case in which the annotation values are $L = \{ 0, 1 \}$. Thus, the classical domain can be specified as $D_{01} = \langle \{ 0, 1 \}, \max, \min, 0, 1 \rangle$. In this case, Annotated RDFS turns out to be the same as standard RDFS.

The Temporal Domain

For our representation of the temporal domain we aim at using non-discrete time as it is necessary to model temporal intervals with any precision. However, for presentation purposes we will show the dates as years only. To start with, *time points* are elements of the value space $\mathbb{Q} \cup \{ -\infty, +\infty \}$ and a *temporal interval* is a non-empty interval $[\alpha_1, \alpha_2]$, where α_i are time points. An empty interval is denoted as $\emptyset$. We define the partial order on intervals as $I_1 \leqslant I_2$ if and only if $I_1 \subseteq I_2$. The intuition here is that if a triple is true at time points in I_2 and $I_1 \leqslant I_2$ then, in particular, it is true at any time point in $I_1 \neq \emptyset$.

Now, apparently the set of intervals would be a candidate for L, however, in order to represent the upper bound interval of $\tau\colon [2001, 2005]$ and $\tau\colon [2008, 2009]$ we rather need the union of intervals, denoted $\{ [2001, 2005], [2008, 2009] \}$, meaning that a triple is true both in the former as well as in the latter interval. Now, we define L as:

$$L = \{ t \mid t \text{ is a finite set of disjoint temporal intervals } \} \cup \{ \bot, \top \} \quad ,$$

where $\bot = \{\emptyset\}, \top = \{[-\infty, +\infty]\}$. Therefore, a *temporal term* is an element $t \in L$, i.e. a set of pairwise disjoint time intervals. We allow the following variations:

(i) $[\alpha]$ as a shorthand for $[\alpha, \alpha]$;
(ii) $\tau\colon \alpha$ as a shorthand of $\tau\colon \{[\alpha]\}$; and
(iii) $\tau\colon [\alpha, \alpha']$ as a shorthand of $\tau\colon \{[\alpha, \alpha']\}$.

Furthermore, on L we define the following partial order:

$$t_1 \preceq t_2 \text{ if and only if } \forall I_1 \in t_1 \exists I_2 \in t_2, \text{ such that } I_1 \leqslant I_2 \quad .$$

Similarly as for time intervals, the intuition for $\preceq$ is that if a triple is true during the time points in all the intervals in t_2 and $t_1 \preceq t_2$, then, in particular, the triple is true at any time point in intervals of t_1. Essentially, if $t_1 \preceq t_2$ then a temporal triple $\tau_2\colon t_2$ is true to a larger "temporal extent" than the temporal triple $\tau_1\colon t_1$.

The partial order $\preceq$ induces the following join ($\oplus$) operation on L. Intuitively, if a triple is true at t_1 and also true at t_2 then it will be true also for time points specified by $t_1 \oplus t_2$ (a kind of union of time points). As an example, if $\tau\colon \{[2002, 2005], [2008, 2010]\}$ and $\tau\colon \{[2004, 2006], [2009, 2012]\}$ are true then we expect that this is the same as saying that $\tau\colon \{[2002, 2006], [2008, 2012]\}$ is true. The join operator will be defined in such a way that $\{[2002, 2005], [2008, 2010]\} \oplus \{[2004, 2006], [2009, 2012]\} = \{[2002, 2006], [2008, 2012]\}$. Operationally, this means that $t_1 \oplus t_2$ will be obtained as follows: (i) take the union of the sets of intervals $t = t_1 \cup t_2$; and (ii) join overlapping intervals in t until no more overlapping intervals can be obtained. Formally,

$$t_1 \oplus t_2 = \text{infimum} \{ t \mid t \succeq t_i, i = 1, 2 \} \quad .$$

It remains to define the $\otimes$ over sets of intervals. Intuitively, we would like to support inferences such that from the triples $(a, \mathsf{sc}, b)\colon \{[2002, 2005], [2008, 2010]\}$ and $(b, \mathsf{sc}, c)\colon \{[2004, 2006], [2009, 2012]\}$, we can infer $(a, \mathsf{sc}, c)\colon \{[2004, 2005], [2009, 2010]\}$, where $\{[2002, 2005], [2008, 2010]\} \otimes \{[2004, 2006], [2009, 2012]\} = \{[2004, 2005], [2009, 2010]\}$. We get it by means of

$$t_1 \otimes t_2 = \text{supremum} \{ t \mid t \preceq t_i, i = 1, 2 \} \quad .$$

Example 6.1 (Temporal domain $\otimes$)**.** Using the following triples regarding another member of the Nightwish band

$$(\texttt{:NightwishMember}, \texttt{sc}, \texttt{mo:MusicArtist}) : [1992, 2012]$$
$$(\texttt{dbpedia:Troy\_Donockley}, \texttt{type}, \texttt{:NightwishMember}) : [1996, 1999] \ ,$$

we can infer that

$$(\texttt{dbpedia:Troy\_Donockley}, \texttt{type}, \texttt{mo:MusicArtist}) : [1996, 1999] \ ,$$

where $\{[1992, 2012]\} \otimes \{[1996, 1999]\} = \{[1996, 1999]\}$.

The Fuzzy Domain

To model fuzzy RDFS (Straccia, 2009) we may define the annotation domain as $D_{[0,1]} = \langle [0, 1], \max, \otimes, 0, 1\rangle$ where $\otimes$ is any continuous t-norm on $[0, 1]$.

Example 6.2 (Fuzzy domain $\otimes$)**.** Adapting our running example to the fuzzy domain we can state the following: Nightwish collaborators are partial members of the band (50%), and since Troy is a Nightwish collaborator:

$$(\texttt{:NightwishCollaborator}, \texttt{sc}, \texttt{:NightwishMember}) : 0.5$$
$$(\texttt{dbpedia:Troy\_Donockley}, \texttt{type}, \texttt{:NightwishCollaborator}) : 0.7$$

Then, e.g. under the product t-norm $\otimes$, we can infer the following triple:

$$(\texttt{dbpedia:Troy\_Donockley}, \texttt{type}, \texttt{:NightwishMember}) : 0.35$$

The Provenance domain

Typically, provenance is identified by a URI, usually the URI of the document in which the triples are defined or possibly a URI identifying a named graph. However, provenance of inferred triples is an issue that has been little tackled in the literature (Delbru et al., 2008; Flouris et al., 2009). The intuition behind our approach is similar to the one of Delbru et al. (2008) and Flouris et al. (2009) where provenance of an inferred triple is defined as the aggregation of provenances of documents that allow to infer that triple.

We start from a countably infinite set of *atomic provenances* $\mathbf{P}$, which in practice can be represented by URIs. We consider the propositional formulae made from symbols in $\mathbf{P}$ (atomic propositions), logical *or* ($\vee$) and logical *and* ($\wedge$), for which we have the standard entailment $\models$. A *provenance value* is an equivalent class for the logical equivalence relation, i.e. the set of annotation values is the quotient set of $\mathbf{P}$ by the logical equivalence. The order relation is $\models$, $\otimes$ and $\oplus$ are $\wedge$ and $\vee$, respectively. We set $\top$ to *true* and $\bot$ to *false*.

Example 6.3 (Provenance domain $\otimes$)**.** Consider the following triples (numbered for easier reference below):

$$(\texttt{dbpedia:Marco\_Hietala}, \texttt{mo:member\_of}, \texttt{:NightwishMember}) : \texttt{dbpedia} \tag{6.1}$$
$$(\texttt{dbpedia:Marco\_Hietala}, \texttt{type}, \texttt{foaf:Person}) : \texttt{dbpedia} \tag{6.2}$$
$$(\texttt{dbpedia:Nightwish}, \texttt{type}, \texttt{mo:MusicGroup}) : \texttt{dbpedia} \tag{6.3}$$
$$(\texttt{foaf:Person}, \texttt{sc}, \texttt{foaf:Agent}) : \texttt{foaf} \tag{6.4}$$
$$(\texttt{mo:member\_of}, \texttt{dom}, \texttt{foaf:Person}) : \texttt{mo} \tag{6.5}$$

$$(\texttt{mo:member\_of}, \mathsf{range}, \texttt{mo:MusicGroup}) : \texttt{mo} \tag{6.6}$$

We can deduce that `dbpedia:Marco_Hietala` is a `foaf:Agent` in two different ways: (a) using statements (6.1), (6.4), and (6.5); or (b) using the statements (6.2) and (6.4). So, it is possible to infer the following annotated triple:

$$(\texttt{dbpedia:Marco\_Hietala}, \mathsf{type}, \texttt{foaf:Agent}) : (\texttt{dbpedia} \wedge \texttt{foaf} \wedge \texttt{mo}) \vee (\texttt{foaf} \wedge \texttt{mo})$$

However, since $(\texttt{dbpedia} \wedge \texttt{foaf} \wedge \texttt{mo}) \vee (\texttt{foaf} \wedge \texttt{mo})$ is logically equivalent to $\texttt{foaf} \wedge \texttt{mo}$, the aggregated inference can be collapsed into:

$$(\texttt{dbpedia:Marco\_Hietala}, \mathsf{type}, \texttt{Agent}) : \texttt{foaf} \wedge \texttt{mo}$$

6.1.5. Deductive system

An important feature of our framework is that we are able to provide a deductive system in the style of the one for classical RDFS. Moreover, *the schemata of the rules are the same for any annotation domain* (only support for the domain dependent $\otimes$ and $\oplus$ operations has to be provided). The rules of our deductive system, as in Section 2.4.3, are arranged in groups that capture the semantic conditions of models, where A, B, C, X, and Y are meta-variables representing elements in **UBL** and D, E represent elements in **UL**. The rule set contains two rules, $(1a)$ and $(1b)$, that are the same as for the crisp case, while rules $(2a)$ to $(5b)$ are the annotated rules homologous to the crisp ones.

1. Simple:

$$(a)\quad \frac{G}{G'} \text{ for an annotated map } \theta : G' \to G \qquad (b)\quad \frac{G}{G'} \text{ for } G' \preceq G$$

2. Subproperty:

$$(a)\quad \frac{(A, \mathsf{sp}, B) : \lambda_1, (B, \mathsf{sp}, C) : \lambda_2}{(A, \mathsf{sp}, C) : \lambda_1 \otimes \lambda_2} \qquad (b)\quad \frac{(D, \mathsf{sp}, E) : \lambda_1, (X, D, Y) : \lambda_2}{(X, E, Y) : \lambda_1 \otimes \lambda_2}$$

3. Subclass:

$$(a)\quad \frac{(A, \mathsf{sc}, B) : \lambda_1, (B, \mathsf{sc}, C) : \lambda_2}{(A, \mathsf{sc}, C) : \lambda_1 \otimes \lambda_2} \qquad (b)\quad \frac{(A, \mathsf{sc}, B) : \lambda_1, (X, \mathsf{type}, A) : \lambda_2}{(X, \mathsf{type}, B) : \lambda_1 \otimes \lambda_2}$$

4. Typing:

$$(a)\quad \frac{(D, \mathsf{dom}, B) : \lambda_1, (X, D, Y) : \lambda_2}{(X, \mathsf{type}, B) : \lambda_1 \otimes \lambda_2} \qquad (b)\quad \frac{(D, \mathsf{range}, B) : \lambda_1, (X, D, Y) : \lambda_2}{(Y, \mathsf{type}, B) : \lambda_1 \otimes \lambda_2}$$

5. Implicit Typing:

$$(a)\quad \frac{(A, \mathsf{dom}, B) : \lambda_1, (D, \mathsf{sp}, A) : \lambda_2, (X, D, Y) : \lambda_3}{(X, \mathsf{type}, B) : \lambda_1 \otimes \lambda_2 \otimes \lambda_3}$$

$$(b)\quad \frac{(A, \mathsf{range}, B) : \lambda_1, (D, \mathsf{sp}, A) : \lambda_2, (X, D, Y) : \lambda_3}{(Y, \mathsf{type}, B) : \lambda_1 \otimes \lambda_2 \otimes \lambda_3}$$

Please note we assume that a rule is not applied if the consequence is of the form $\tau : \bot$ (see Section 6.1.3).

Like for the classical case, the *closure* is defined as $cl(G) = \{\, \tau : \lambda \mid G \vdash^* \tau : \lambda \,\}$, where $\vdash^*$ is as $\vdash$ without rule $(1a)$. Note again that the size of the closure of G is polynomial in $|G|$ and can be computed in polynomial time, provided that the computational complexity of operations $\otimes$ and $\oplus$ are polynomially bounded (from a computational complexity point of view, it is as for the classical case, plus the cost of the operations $\otimes$ and $\oplus$ in L).

Furthermore note that $cl(G)$ is not guaranteed to be a normalised annotated RDF graph. In order to ensure a normalised graph we can apply the following rule, denoted *generalisation rule*:

$$\frac{(X, A, Y) : \lambda_1, (X, A, Y) : \lambda_2}{(X, A, Y) : \lambda_1 \oplus \lambda_2} \ ,$$

where each application of this rule removes the premises from the graph. Let us show an example of the application of the generalisation rule.

Example 6.4 (Generalisation operation)**.** Consider the following triples along with our running example from Data 6.1:

$$(\texttt{foaf:name}, \mathsf{dom}, \texttt{foaf:Person})\colon [-\infty, +\infty]^a$$
$$(\texttt{foaf:Person}, \mathsf{sc}, \texttt{foaf:Agent})\colon [-\infty, +\infty]$$
$$(\texttt{mo:MusicArtist}, \mathsf{sc}, \texttt{foaf:Agent})\colon [-\infty, +\infty]\ ,$$

we infer the following triples:

$$(\texttt{dbpedia:Marco\_Hietala}, \mathsf{type}, \texttt{foaf:Agent})\colon [1984, 2012]$$
$$(\texttt{dbpedia:Marco\_Hietala}, \mathsf{type}, \texttt{foaf:Agent})\colon [1966, 2012]\ .$$

The application of the "Generalisation" rule will collapse the triples:

$$(\texttt{dbpedia:Marco\_Hietala}, \mathsf{type}, \texttt{foaf:Agent})\colon [1966, 2012]\ ,$$

since $[1984, 2012] \oplus [1966, 2012] = [1966, 2012]$.

[a] We assume this triple is valid to reduce the number of triples shown, the domain of `foaf:name` is in fact `owl:Thing`.

Proposition 6.1 (Soundness and completeness)**.** *For two annotated graphs G and G', the proof system $\vdash$ is sound and complete for $\models$, that is $G \vdash G'$ if and only if $G \models G'$.*

Proof: [Extends (Muñoz et al., 2009)] ($\Rightarrow$) Let $\mathcal{I} = \langle \Delta_R, \Delta_P, \Delta_C, \Delta_L, P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}} \rangle$ be an interpretation such that $\mathcal{I} \models G$ i.e. $\mathcal{I}$ satisfies all the conditions of Definition 6.5. Furthermore let $\tau\colon \lambda$ be the result from an instantiation of a rule $2-5$, such that $\frac{R}{\tau\colon \lambda}$, where $R \subseteq G$ and let $G' = G \cup \{\, \tau\colon \lambda \,\}$, then $\mathcal{I} \models G'$.

1. Simple:

(a) We show that if $G' \to G$ then $G \models G'$. Let θ be an annotated map such that $\theta(G') \preceq G$. Consider the function $A'\colon \mathbf{B} \to \Delta_R$ such that

$$A'(x) = \begin{cases} A(\theta(x)) \text{ if } \theta(x) \in \mathbf{B} \\ \theta(x)^{\mathcal{I}} \text{ if } \theta(x) \notin \mathbf{B} \end{cases} .$$

Note that: a) if $x \in \mathbf{B}$ and $\theta(x) \in \mathbf{B}$ we get that $\theta(x)^{\mathcal{I}_{\mathcal{A}}} = A(\theta(x)) = A'(x) = x^{\mathcal{I}_{\mathcal{A}'}}$; b) if $x \in \mathbf{B}$ and $\theta(x) \notin \mathbf{B}$ we get that $\theta(x)^{\mathcal{I}_{\mathcal{A}}} = \theta(x)^{\mathcal{I}} = A'(x) = x^{\mathcal{I}_{\mathcal{A}'}}$; c) if $x \notin \mathbf{B}$ we get that $\theta(x) = x$ and $\theta(x)^{\mathcal{I}_{\mathcal{A}}} = x^{\mathcal{I}} = A'(x) = x^{\mathcal{I}_{\mathcal{A}'}}$. Thus we have that $\theta(x)^{\mathcal{I}_{\mathcal{A}}} = x^{\mathcal{I}_{\mathcal{A}'}}$ for all $x \in \mathbf{UB}$. Let $(s, p, o)\colon \lambda \in G'$, then $(\theta(s), \theta(p), \theta(o))\colon \lambda = (\theta(s), p, \theta(o))\colon \lambda \in G$. Since $\mathcal{I} \models G$ we have that $p^{\mathcal{I}} \in \Delta_P$ and $P[\![\theta(s)^{\mathcal{I}_{\mathcal{A}}}, \theta(s)^{\mathcal{I}_{\mathcal{A}}}]\!] \succeq \lambda$ and $P[\![\theta(s)^{\mathcal{I}_{\mathcal{A}'}}, \theta(s)^{\mathcal{I}_{\mathcal{A}'}}]\!] \succeq \lambda$. Thus $\mathcal{I}$ satisfies condition (1) from Definition 6.5 for G' (with function A') and satisfies all other conditions of Definition 6.5, so $\mathcal{I} \models G'$.

(b) if $G' \preceq G$ then $G' \to G$ and thus $G \models G'$.

2. Subproperty:

(a) Let $\mathcal{I} \models (A, \mathsf{sp}, B)\colon \lambda_1$ and $\mathcal{I} \models (B, \mathsf{sp}, C)\colon \lambda_2$. It follows that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![\mathsf{sp}^{\mathcal{I}}]\!](B^{\mathcal{I}}, C^{\mathcal{I}}) \succeq \lambda_2$. According to condition "Subproperty 1." from Definition 6.5, we have that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![\mathsf{sp}^{\mathcal{I}}]\!](B^{\mathcal{I}}, C^{\mathcal{I}}) \preceq P[\![\mathsf{sp}^{\mathcal{I}}]\!](A^{\mathcal{I}}, C^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq P[\![\mathsf{sp}^{\mathcal{I}}]\!](A^{\mathcal{I}}, C^{\mathcal{I}})$. Therefore $\mathcal{I} \models (A, \mathsf{sp}, C)\colon \lambda_1 \otimes \lambda_2$.

(b) Let $\mathcal{I} \models (D, \mathsf{sp}, E)\colon \lambda_1$ and $\mathcal{I} \models (X, D, Y)\colon \lambda_2$. It follows that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, E^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \succeq \lambda_2$. According to condition "Subproperty 2." from Definition 6.5, we have that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, E^{\mathcal{I}}) \otimes P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq P[\![E^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq P[\![E^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$. Therefore $\mathcal{I} \models (X, E, Y)\colon \lambda_1 \otimes \lambda_2$.

3. Subclass:

(a) Let $\mathcal{I} \models (A, \mathsf{sc}, B) \colon \lambda_1$ and $\mathcal{I} \models (B, \mathsf{sc}, C) \colon \lambda_2$. It follows that $P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![\mathsf{sc}^{\mathcal{I}}]\!](B^{\mathcal{I}}, C^{\mathcal{I}}) \succeq \lambda_2$. According to condition "Subclass 1." from Definition 6.5, we have that $P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![\mathsf{sc}^{\mathcal{I}}]\!](B^{\mathcal{I}}, C^{\mathcal{I}}) \preceq P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, C^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, C^{\mathcal{I}})$. Therefore $\mathcal{I} \models (A, \mathsf{sc}, C) \colon \lambda_1 \otimes \lambda_2$.

(b) Let $\mathcal{I} \models (A, \mathsf{sc}, B) \colon \lambda_1$ and $\mathcal{I} \models (X, \mathsf{type}, A) \colon \lambda_2$. It follows that $P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![\mathsf{type}^{\mathcal{I}}]\!](X^{\mathcal{I}}, A^{\mathcal{I}}) \succeq \lambda_2$. From condition "Typing I, 1." (Definition 6.5), we have that $C[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}) \succeq \lambda_2$. According to condition "Subclass 2." from Definition 6.5, we have that $P[\![\mathsf{sc}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \otimes C[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}) \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$ and, again from condition "Typing I, 1.", $\lambda_1 \otimes \lambda_2 \preceq P[\![\mathsf{type}^{\mathcal{I}}]\!](X^{\mathcal{I}}, B^{\mathcal{I}})$. Therefore $\mathcal{I} \models (X, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2$.

4. Typing:

(a) Let $\mathcal{I} \models (D, \mathsf{dom}, B) \colon \lambda_1$ and $\mathcal{I} \models (X, D, Y) \colon \lambda_2$. It follows that $P[\![\mathsf{dom}^{\mathcal{I}}]\!](D^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \succeq \lambda_2$. From condition "Typing I, 2." (Definition 6.5), we have that $P[\![\mathsf{dom}^{\mathcal{I}}]\!](D^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$. From condition "Typing I, 1." (Definition 6.5), we have that $\lambda_1 \otimes \lambda_2 \preceq P[\![\mathsf{type}^{\mathcal{I}}]\!](X^{\mathcal{I}}, B^{\mathcal{I}})$. Therefore $\mathcal{I} \models (X, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2$.

(b) Let $\mathcal{I} \models (D, \mathsf{range}, B) \colon \lambda_1$ and $\mathcal{I} \models (X, D, Y) \colon \lambda_2$. It follows that $P[\![\mathsf{range}^{\mathcal{I}}]\!](D^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$ and $P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \succeq \lambda_2$. From condition "Typing I, 3." (Definition 6.5), we have that $P[\![\mathsf{range}^{\mathcal{I}}]\!](D^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq C[\![B^{\mathcal{I}}]\!](Y^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq C[\![B^{\mathcal{I}}]\!](Y^{\mathcal{I}})$. From condition "Typing I, 1." (Definition 6.5), we have that $\lambda_1 \otimes \lambda_2 \preceq P[\![\mathsf{type}^{\mathcal{I}}]\!](Y^{\mathcal{I}}, B^{\mathcal{I}})$. Therefore $\mathcal{I} \models (Y, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2$.

5. Implicit Typing:

(a) Let $\mathcal{I} \models (A, \mathsf{dom}, B) \colon \lambda_1$, $\mathcal{I} \models (D, \mathsf{sp}, A) \colon \lambda_2$, and $\mathcal{I} \models (X, D, Y) \colon \lambda_3$. It follows that $P[\![\mathsf{dom}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$, $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, A^{\mathcal{I}}) \succeq \lambda_2$, and $P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \succeq \lambda_3$. According to condition "Subproperty 2." from Definition 6.5, we have that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, A^{\mathcal{I}}) \otimes P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$. From condition "Typing I, 2." (Definition 6.5), we have that $P[\![\mathsf{dom}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \otimes \lambda_3 \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$. From condition "Typing I, 1." (Definition 6.5), we have that $\lambda_1 \otimes \lambda_2 \otimes \lambda_3 \preceq P[\![\mathsf{type}^{\mathcal{I}}]\!](X^{\mathcal{I}}, B^{\mathcal{I}})$. Therefore $\mathcal{I} \models (X, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2 \otimes \lambda_3$.

(b) Let $\mathcal{I} \models (A, \mathsf{range}, B) \colon \lambda_1$, $\mathcal{I} \models (D, \mathsf{sp}, A) \colon \lambda_2$, and $\mathcal{I} \models (X, D, Y) \colon \lambda_3$. It follows that $P[\![\mathsf{range}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \succeq \lambda_1$, $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, A^{\mathcal{I}}) \succeq \lambda_2$, and $P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \succeq \lambda_3$. According to condition "Subproperty 2." from Definition 6.5, we have that $P[\![\mathsf{sp}^{\mathcal{I}}]\!](D^{\mathcal{I}}, A^{\mathcal{I}}) \otimes P[\![D^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \preceq P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}})$. From condition "Typing I, 3." (Definition 6.5), we have that $P[\![\mathsf{range}^{\mathcal{I}}]\!](A^{\mathcal{I}}, B^{\mathcal{I}}) \otimes P[\![A^{\mathcal{I}}]\!](X^{\mathcal{I}}, Y^{\mathcal{I}}) \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$ and thus $\lambda_1 \otimes \lambda_2 \otimes \lambda_3 \preceq C[\![B^{\mathcal{I}}]\!](X^{\mathcal{I}})$. From condition "Typing I, 1." (Definition 6.5), we have that $\lambda_1 \otimes \lambda_2 \otimes \lambda_3 \preceq P[\![\mathsf{type}^{\mathcal{I}}]\!](X^{\mathcal{I}}, B^{\mathcal{I}})$. Therefore $\mathcal{I} \models (X, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2 \otimes \lambda_3$.

($\Leftarrow$) Given an annotated graph G, let $\mathcal{I} = \langle \Delta_R, \Delta_P, \Delta_C, \Delta_L, P[\![\cdot]\!], C[\![\cdot]\!], \cdot^{\mathcal{I}} \rangle$ be an interpretation defined as follows:

- $\Delta_R = \mathit{universe}(G) \cup \rho\mathrm{df}$;
- $\Delta_P = \{\, p \in \mathit{voc}(G) \mid (s, p, o) \colon \lambda \in \mathit{cl}(G) \,\} \cup$
  $\rho\mathrm{df} \cup \{\, p \in \mathit{universe}(G) \mid (p, \mathsf{sp}, x) \colon \lambda, (y, \mathsf{sp}, p) \colon \lambda, (p, \mathsf{dom}, z) \colon \lambda \text{ or } (p, \mathsf{range}, v) \colon \lambda \in G \,\}$;
- $\Delta_C = \{\, c \in \mathit{universe}(G) \mid (x, \mathsf{type}, c) \colon \lambda \in G \,\} \cup$
  $\{\, c \in \mathit{universe}(G) \mid (c, \mathsf{sc}, x) \colon \lambda, (y, \mathsf{sc}, c) \colon \lambda, (z, \mathsf{dom}, c) \colon \lambda \text{ or } (v, \mathsf{range}, c) \colon \lambda \in G \,\}$;
- $\Delta_L = \mathbf{L} \cap \mathit{universe}(G)$;
- $P[\![\cdot]\!] : \Delta_P \to 2^{\Delta_R \times \Delta_R}$ is an interpretation function such that:
  - if $p \in \mathbf{U} \cap \Delta_P$ and $(x, p, y) \colon \lambda \in \mathit{cl}(G)$ then $P[\![p]\!](x, y) \succeq \lambda$;

– if $p \in \mathbf{B} \cap \Delta_P$ and $(p, \mathsf{sp}, p') \colon \lambda_1, (x, p', y) \colon \lambda_2 \in cl(G)$ then $P[\![p]\!](x, y) \succeq \lambda_1 \otimes \lambda_2$ such that $\lambda_1 \otimes \lambda_2 \neq \bot$.

- $C[\![\cdot]\!] : \Delta_C \to L$ is an interpretation function such that $(x, \mathsf{type}, c) \colon \lambda \in cl(G)$, $C[\![c]\!](x) \succeq \lambda$;
- $\cdot^{\mathcal{I}}$ is the identity function over $universe(G) \cup \rho\text{df}$.

We have that $\mathcal{I} \models G$ if $\mathcal{I}$ satisfies all the conditions from Definition 6.5:

Simple:

(a) First note that from the construction of $cl(G)$, $universe(cl(G)) = universe(G) \cup \rho\text{df}$. Let $(s, p, o) \colon \lambda \in G$ then, from the construction of $\mathcal{I}$, we have that $p^{\mathcal{I}} = p \in \Delta_P$ and $P[\![p]\!]\big(s^{\mathcal{I}}, o^{\mathcal{I}}\big) = \lambda$ and thus $\mathcal{I}$ satisfies condition (i) for G.

(b) We now show that if $G \models G'$ then there is an annotated map $G' \to cl(G)$. From the construction of $\mathcal{I}$ we have that $\mathcal{I} \models G$ and since $G \models G'$, $\mathcal{I} \models G'$. Since $\mathcal{I}$ satisfies condition (i) there exists a function $A \colon \mathbf{B} \to universe(G) \cup \rho\text{df}$ such that for each $(s, p, o) \colon \lambda \in G'$, $p \in \Delta_P$ and $P[\![p^{\mathcal{I}}]\!]\big(s^{\mathcal{I}_{\mathcal{A}}}, o^{\mathcal{I}_{\mathcal{A}}}\big) = \lambda$. Since $p \in \mathbf{U}$, we have that $p^{\mathcal{I}} = p^{\mathcal{I}_{\mathcal{A}}} = p$ and thus $P[\![p^{\mathcal{I}}]\!]\big(s^{\mathcal{I}_{\mathcal{A}}}, o^{\mathcal{I}_{\mathcal{A}}}\big) = P[\![p]\!]\big(s^{\mathcal{I}_{\mathcal{A}}}, o^{\mathcal{I}_{\mathcal{A}}}\big) = \lambda$ for each $(s, p, o) \colon \lambda \in cl(G)$. Since $P[\![p^{\mathcal{I}}]\!]\big(s^{\mathcal{I}_{\mathcal{A}}}, o^{\mathcal{I}_{\mathcal{A}}}\big) = \lambda$, we have that $(s^{\mathcal{I}_{\mathcal{A}}}, p^{\mathcal{I}_{\mathcal{A}}}, o^{\mathcal{I}_{\mathcal{A}}}) \colon \lambda \in cl(G)$ for each $(s, p, o) \colon \lambda \in G'$. Thus $\mathcal{I}_{\mathcal{A}} \colon G' \to cl(G)$ is an annotated map $G' \to cl(G)$.

Subproperty:

(a) Let $P[\![\mathsf{sp}^{\mathcal{I}}]\!]\big(A^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1$ and $P[\![\mathsf{sp}^{\mathcal{I}}]\!]\big(B^{\mathcal{I}}, C^{\mathcal{I}}\big) \succeq \lambda_2$. From the construction of $\mathcal{I}$ we have that $(A, \mathsf{sp}, B) \colon \lambda_1, (B, \mathsf{sp}, C) \colon \lambda_2 \in cl(G)$ and $A, B, C \in \Delta_P$. Since $cl(G)$ is closed under application of rule $(2a)$ we have that $(A, \mathsf{sp}, C) \colon \lambda_1 \otimes \lambda_2 \in cl(G)$ and thus $P[\![\mathsf{sp}^{\mathcal{I}}]\!]\big(A^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

(b) Let $P[\![D^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, Y^{\mathcal{I}}\big) \succeq \lambda_1$ and $P[\![\mathsf{sp}^{\mathcal{I}}]\!]\big(D^{\mathcal{I}}, E^{\mathcal{I}}\big) \succeq \lambda_2$, thus $(D, \mathsf{sp}, E) \colon \lambda_2 \in cl(G)$ and $D, E \in \Delta_P$. We must consider the following cases:

- if $D \in \mathbf{U}$ then, from the construction of $\mathcal{I}$ we have that $(X, D, Y) \colon \lambda_1 \in cl(G)$. If $E \in \mathbf{U}$ and since $cl(G)$ is closed under the application of rule $(2b)$, we also have that $(X, E, Y) \colon \lambda_1 \otimes \lambda_2 \in cl(G)$. Therefore $P[\![E^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, Y^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$. If $E \in \mathbf{B}$, then $(X, D, Y) \colon \lambda_1, (D, \mathsf{sp}, E) \colon \lambda_2 \in cl(G)$, and from the construction of $\mathcal{I}$ we have that $P[\![E^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, Y^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.
- if $D \in \mathbf{B}$ by the construction of $\mathcal{I}$ there exists D' such that $(D', \mathsf{sp}, D) \colon \lambda_3, (X, D', Y) \colon \lambda_4 \in cl(G)$ and $D' \in \Delta_P$. Since $cl(G)$ is closed under the application of rule $(2a)$, we also have that $(D', \mathsf{sp}, E) \colon \lambda_2 \otimes \lambda_3 \in cl(G)$. If $E \in \mathbf{U}$ as $(D', \mathsf{sp}, E) \colon \lambda_2 \otimes \lambda_3, (X, D', Y) \colon \lambda_4 \in cl(G)$ and since $cl(G)$ is closed under the application of rule $(2b)$, we also have that $(X, E, Y) \colon \lambda_2 \otimes \lambda_3 \otimes \lambda_4 \in cl(G)$. Therefore, from the construction of $\mathcal{I}$, we have that $P[\![E^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, Y^{\mathcal{I}}\big) \succeq \lambda_2 \otimes \lambda_3 \otimes \lambda_4$. If $E \in \mathbf{B}$, then $(X, D, Y) \colon \lambda_1, (D, \mathsf{sp}, E) \colon \lambda_2 \in cl(G)$, and from the construction of $\mathcal{I}$ we have that $P[\![E^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, Y^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

Subclass:

(a) Let $P[\![\mathsf{sc}^{\mathcal{I}}]\!]\big(A^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1$ and $P[\![\mathsf{sc}^{\mathcal{I}}]\!]\big(B^{\mathcal{I}}, C^{\mathcal{I}}\big) \succeq \lambda_2$. From the construction of $\mathcal{I}$ we have that $(A, \mathsf{sc}, B) \colon \lambda_1, (B, \mathsf{sc}, C) \colon \lambda_2 \in cl(G)$ and $A, B, C \in \Delta_C$. Since $cl(G)$ is closed under application of rule $(3a)$ we have that $(A, \mathsf{sc}, C) \colon \lambda_1 \otimes \lambda_2 \in cl(G)$ and thus $P[\![\mathsf{sc}^{\mathcal{I}}]\!]\big(A^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

(b) Let $C[\![A^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}\big) \succeq \lambda_1$ and $P[\![\mathsf{sc}^{\mathcal{I}}]\!]\big(A^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_2$. From condition "Typing I, 1." (Definition 6.5), we have that $P[\![\mathsf{type}^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, A^{\mathcal{I}}\big) = \lambda_2$ and thus $\mathcal{I} \models (X, \mathsf{type}, A) \colon \lambda_2$. Since $cl(G)$ is closed under application of rule $(3b)$ we have that $(X, \mathsf{type}, B) \colon \lambda_1 \otimes \lambda_2$. Then $P[\![\mathsf{type}^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$ and thus $C[\![B^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

Typing I:

(a) Let $P[\![\mathsf{type}^{\mathcal{I}}]\!](X, C) \succeq \lambda_1$, by construction of $\mathcal{I}$ we have that $C \in \Delta_C$ and $(X, \mathsf{type}, C) \colon \lambda_1 \in cl(G)$. Also by the construction of $C[\![\cdot]\!]$ we have that $C[\![C^{\mathcal{I}}]\!](X) \succeq \lambda_1$. On the other hand,

if $C \in \Delta_C$ and $C[\![C^{\mathcal{I}}]\!](X) \succeq \lambda_1$, by construction of $C[\![\cdot]\!]$ we have that $(X, \mathsf{type}, C)\colon \lambda_1 \in cl(G)$ and so $P[\![\mathsf{type}^{\mathcal{I}}]\!](X, C) \succeq \lambda_1$.

(b) Let $P[\![\mathsf{dom}^{\mathcal{I}}]\!](D, B) \succeq \lambda_1$ and $P[\![D]\!](X, Y) \succeq \lambda_2$. By construction of $\mathcal{I}$ we have that $D \in \Delta_P$ and $B \in \Delta_C$. Since $cl(G)$ is closed under application of rule (4*a*) we have that $(X, \mathsf{type}, B)\colon \lambda_1 \otimes \lambda_2 \in cl(G)$. Then $P[\![\mathsf{type}]\!]\big(X^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$ and $C[\![B^{\mathcal{I}}]\!]\big(X^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

(c) Let $P[\![\mathsf{range}^{\mathcal{I}}]\!](D, B) \succeq \lambda_1$ and $P[\![D]\!](X, Y) \succeq \lambda_2$. By construction of $\mathcal{I}$ we have that $D \in \Delta_P$ and $B \in \Delta_C$. Since $cl(G)$ is closed under application of rule (4*b*) we have that $(Y, \mathsf{type}, B)\colon \lambda_1 \otimes \lambda_2 \in cl(G)$. Then $P[\![\mathsf{type}]\!]\big(Y^{\mathcal{I}}, B^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$ and $C[\![B^{\mathcal{I}}]\!]\big(Y^{\mathcal{I}}\big) \succeq \lambda_1 \otimes \lambda_2$.

Typing II: The definition of Δ_R and Δ_P satisfy all of these conditions.

□

6.1.6. Query Answering

Informally, queries are as for the classical case where triples are replaced with annotated triples in which *annotation variables* (taken from an appropriate alphabet and denoted Λ) may occur. We allow built-in triples of the form (s, p, o), where p is a built-in predicate taken from a reserved vocabulary and having a *fixed interpretation* on the annotation domain D, such as $(\lambda, \preceq, l)$ stating that the value of λ has to be $\preceq$ than the value $l \in L$. We generalise the built-ins to any n-ary predicate p, where p's arguments may be annotation variables, ρdf variables, domain values of D, values from **UL**, and p has a fixed interpretation. We will assume that the evaluation of the predicate can be decided in finite time. As for the crisp case, for convenience, we write "functional predicates" as *assignments* of the form $x{:=}f(\bar{\mathbf{z}})$ and assume that the function $f(\bar{\mathbf{z}})$ is safe. Furthermore, we also assume that any non functional built-in predicate $p(\bar{\mathbf{z}})$ should be safe as well.

For instance, informally for a given time interval $[t_1, t_2]$, we may define $x{:=}\mathit{length}([t_1, t_2])$ as true if and only if the value of x is $t_2 - t_1$.

Example 6.5 (Annotated query)**.** Considering our dataset from Data 6.1 as input and the query asking for artists that were members of the Nightwish band between 2000 and 2010 and the temporal term at which this was true:

$$q(x, \Lambda) \leftarrow (\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, x)\colon \Lambda', \Lambda{:=}(\Lambda' \wedge [2000, 2010])$$

will get the following answers:

$$\langle \texttt{dbpedia:Marco\_Hietala}, [2001, 2010] \rangle$$
$$\langle \texttt{dbpedia:Tarja\_Turunen}, [2000, 2005] \rangle \ .$$

Formally, an *annotated query* is of the form

$$q(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}) \leftarrow \exists \bar{\mathbf{y}} \exists \bar{\mathbf{\Lambda}}'.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}')$$

in which $\varphi(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}')$ is a conjunction (as for the crisp case, we use '**,**' as conjunction symbol) of annotated triples and built-in predicates, $\bar{\mathbf{x}}$ and $\bar{\mathbf{\Lambda}}$ are the distinguished variables, $\bar{\mathbf{y}}$ and $\bar{\mathbf{\Lambda}}'$ are the *non-distinguished variables* (existential quantified variables), and $\bar{\mathbf{x}}$, $\bar{\mathbf{\Lambda}}$, $\bar{\mathbf{y}}$ and $\bar{\mathbf{\Lambda}}'$ are pairwise disjoint. Variables in $\bar{\mathbf{\Lambda}}$ and $\bar{\mathbf{\Lambda}}'$ can only appear in annotations or built-in predicates and furthermore, the query head must contain at least one variable.

Given an annotated graph G, a query $q(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}) \leftarrow \exists \bar{\mathbf{y}} \exists \bar{\mathbf{\Lambda}}'.\varphi(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}')$, a vector $\bar{\mathbf{t}}$ of terms in *universe*(G) and a vector $\bar{\lambda}$ of annotated terms in L, we say that $q(\bar{\mathbf{t}}, \bar{\lambda})$ is *entailed* by G, denoted $G \models q(\bar{\mathbf{t}}, \bar{\lambda})$, if and only if in any model $\mathcal{I}$ of G, there is a vector $\bar{\mathbf{t}}'$ of terms in *universe*(G) and a vector $\bar{\lambda}'$ of annotation

values in L such that $\mathcal{I}$ is a model of $\varphi(\bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{t}}', \bar{\lambda}')$. If $G \models q(\bar{\mathbf{t}}, \bar{\lambda})$ then $\langle \bar{\mathbf{t}}, \bar{\lambda} \rangle$ is called an *answer* to q. The *answer set* of q w.r.t. G is ($\preceq$ extends to vectors point-wise)

$$ans(G, q) = \{\ \langle \bar{\mathbf{t}}, \bar{\lambda} \rangle \mid G \models q(\bar{\mathbf{t}}, \bar{\lambda}), \bar{\lambda} \neq \bar{\bot} \text{ and for any } \bar{\lambda}' \neq \bar{\lambda} \text{ such that } G \models q(\bar{\mathbf{t}}, \bar{\lambda}'), \bar{\lambda}' \preceq \bar{\lambda} \text{ holds } \}\ .$$

That is, for any tuple $\bar{\mathbf{t}}$, the vector of annotation values $\bar{\lambda}$ is as large as possible. This is to avoid that redundant/subsumed answers occur in the answer set. The following can be shown:

Proposition 6.2. *Given a graph G, $\langle \bar{\mathbf{t}}, \bar{\lambda} \rangle$ is an* answer *to q if and only if $\exists \bar{\mathbf{y}} \exists \bar{\mathbf{\Lambda}}'.\varphi(\bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}')$ is true in the closure of G and λ is $\preceq$-maximal.*[6]

Queries with Aggregates

Next we extend the query language by allowing so-called aggregates to occur in a query. Essentially, aggregates may be like the usual SQL aggregate functions such as `SUM`, `AVG`, `MAX`, `MIN`. But, we have also domain specific aggregates such as $\oplus$ and $\otimes$. The following examples present some queries that can be expressed with the use of built-in queries and aggregates.

Example 6.6 (Assignment query)**.** Using a built-in aggregate we can pose a query that, for each band member, retrieves his maximal time of employment for any band in the following way:

$$q(x, maxL) \leftarrow (y, \texttt{foaf:member}, x) \colon \lambda, maxL{:=}maxlength(\lambda)\ .$$

Here, the *maxlength* built-in predicate returns, given a set of temporal intervals, the maximal interval in the set.

Example 6.7 (Aggregation query)**.** Suppose we are looking for artists that are members of some `mo:MusicGroup` for a certain time period and we would like to know the average length of their membership. Then such a query will be expressed as

$$q(x, avgL) \leftarrow\ (y, \texttt{foaf:member}, x) \colon \lambda, \mathsf{GroupedBy}(x)\,, avgL{:=}\texttt{AVG}[length(\lambda)]\ .$$

Essentially, we group by the artist, compute for each artist the time he was a member of the `mo:MusicGroup` (by means of the built-in function *length*), and finally compute the average value for each group. That is, $g = \{\langle t, t_1 \rangle, \ldots, \langle t, t_n \rangle\}$ is a group of tuples with the same value t for artist x, and value t_i for y, where each length of membership for t_i is l_i (computed as $length(\cdot)$), then the value of $avgL$ for the group g is $(\sum_i l_i)/n$.

Formally, let @ be an aggregate function with @ $\in \{\texttt{SUM}, \texttt{AVG}, \texttt{MAX}, \texttt{MIN}, \texttt{COUNT}, \oplus, \otimes\}$ then a query with aggregates is of the form

$$\begin{aligned} q\big(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \alpha\big) \leftarrow\ & \exists \bar{\mathbf{y}} \exists \bar{\mathbf{\Lambda}}'.\varphi\big(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}'\big)\,, \\ & \mathsf{GroupedBy}(\bar{\mathbf{w}})\,, \\ & \alpha{:=}@[f(\bar{\mathbf{z}})] \end{aligned}$$

where $\bar{\mathbf{w}}$ are variables in $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$ or $\bar{\mathbf{\Lambda}}$, each variable in $\bar{\mathbf{x}}$ and $\bar{\mathbf{\Lambda}}$ must occur in $\bar{\mathbf{w}}$ and any variable in $\bar{\mathbf{z}}$ occurs in $\bar{\mathbf{y}}$ or $\bar{\mathbf{\Lambda}}'$. From a semantics point of view, we say that $\mathcal{I}$ *is a model of* (*satisfies*) $q(\bar{\mathbf{t}}, \bar{\lambda}, a)$, denoted $\mathcal{I} \models q(\bar{\mathbf{t}}, \bar{\lambda}, a)$ if and only if $a = @[a_1, \ldots, a_k]$, where $g = \{\langle \bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{t}}'_1, \bar{\lambda}'_1 \rangle, \ldots, \langle \bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{t}}'_k, \bar{\lambda}'_k \rangle\}$ is a group of k tuples with identical projection on the variables in $\bar{\mathbf{w}}$, $\varphi(\bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{t}}'_r, \bar{\lambda}'_r)$ is true in $\mathcal{I}$ and $a_r = f(\bar{\bar{\mathbf{t}}})$ where $\bar{\bar{\mathbf{t}}}$

[6] $\exists \bar{\mathbf{y}} \exists \bar{\mathbf{\Lambda}}'.\varphi(\bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}')$ is true in the closure of G if and only if for some $\bar{\mathbf{t}}'$, $\bar{\lambda}'$ for all triples in $\varphi(\bar{\mathbf{t}}, \bar{\lambda}, \bar{\mathbf{t}}', \bar{\lambda}')$ there is a triple in $cl(G)$ that subsumes it and the built-in predicates are true, where an annotated triple $\tau \colon \lambda_1$ subsumes $\tau \colon \lambda_2$ if and only if $\lambda_2 \preceq \lambda_1$.

is the projection of $\langle \bar{\mathbf{t}}'_r, \bar{\lambda}'_r \rangle$ on the variables $\bar{\mathbf{z}}$. Now, the notion of $G \models q(\bar{\mathbf{t}}, \bar{\lambda}, a)$ is as usual, any model of G is a model of $q(\bar{\mathbf{t}}, \bar{\lambda}, a)$.

Eventually, we further allow to order answers according to some ordering functions.

Example 6.8 (Ordering query)**.** Consider Example 6.7. We additionally would like to order the artists according to the average length of membership to a band. Then such a query will be expressed as:

$$q(x, avgL) \leftarrow (y, \texttt{foaf:member}, x) \colon \lambda, \mathsf{GroupedBy}(x)\,,$$
$$avgL := \texttt{AVG}[length(\lambda)], \mathsf{OrderBy}(avgL) \quad .$$

Formally, a query with ordering is of the form

$$q(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, z) \quad \leftarrow \quad \exists \bar{\mathbf{y}} \exists \mathbf{\Lambda}'. \varphi(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}'), \mathsf{OrderBy}(z)$$

or, in case grouping is allowed as well, it is of the form

$$\begin{aligned} q(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, z, \alpha) \quad \leftarrow \quad & \exists \bar{\mathbf{y}} \exists \mathbf{\Lambda}'. \varphi(\bar{\mathbf{x}}, \bar{\mathbf{\Lambda}}, \bar{\mathbf{y}}, \bar{\mathbf{\Lambda}}'), \\ & \mathsf{GroupedBy}(\bar{\mathbf{w}}), \\ & \alpha := @[f(\bar{\mathbf{z}})], \\ & \mathsf{OrderBy}(z) \end{aligned}$$

From a semantics point of view, the notion of $G \models q(\bar{\mathbf{t}}, \bar{\lambda}, z, a)$ is as before, but the notion of answer set has to be enforced with the fact that the answers are now ordered according to the assignment of the variable z. Of course, we require that the set of values over which z ranges can be ordered (like string, integers, reals). In case the variable z is an annotation variable, the order is induced by $\preceq$.

6.2. AnQL: Annotated SPARQL

The query language introduced so far allows for conjunctive queries. Languages like SQL and SPARQL allow to pose more complex queries including built-in predicates to filter solutions and advanced features such as negation or aggregates. In this section we will present an extension of the SPARQL (Prud'hommeaux and Seaborne, 2008) query language, called AnQL, that enables querying annotated graphs.

For the rest of this section we fix a specific annotation domain, $D = \langle L, \oplus, \otimes, \bot, \top \rangle$, as defined in Section 6.1.2.

6.2.1. Syntax

A *simple AnQL query* is defined – analogously to a SPARQL query in Section 3.3 – as a quadruple $Q = (P, G, V, A)$ with the differences that:

(1) G is an annotated RDF graph;
(2) we allow annotated graph patterns as presented in Definition 6.6; and
(3) A is the set of annotation variables taken from an infinite set $\mathbf{A}$ (distinct from $\mathbf{V}$).

We now introduce the definition of *Annotated Graph Patterns*:

Definition 6.6 (Annotated Graph Patterns)**.** *Let* $\mathbf{U}$*,* $\mathbf{B}$*,* $\mathbf{L}$*, and* $\mathbf{V}$ *be defined as before. Furthermore, let* λ *be an annotation value from* L *or an annotation variable from* $\mathbf{A}$*, we call* λ *an* annotation label*. An* annotated graph pattern *in AnQL is defined (similar to SPARQL) inductively as follows:*

- *for a triple pattern* τ*,* $\tau \colon \lambda$ *(called an* annotated triple pattern*) is an annotated graph pattern;*

- *a set of annotated triple patterns, called a Basic Annotated Pattern (BAP), is an annotated graph pattern;*
- *if P and P' are annotated graph patterns, then $(P$ `and` $P')$, $(P$ `optional` $P')$, $(P$ `union` $P')$ are annotated graph patterns;*
- *if P is an annotated graph pattern and R is a filter expression, then $(P$ `filter` $R)$ is an annotated graph pattern.*

We further denote by $avars(P)$ the set of annotation variables present in a graph pattern P and $vars(P)$ is extended to include also the annotation variables.

The `optional` operator in the annotated case may cause the values of annotation variables outside the `optional` to change depending if the optional pattern is matched. This is presented in Example 6.9. Please note that in query examples we will use a simple extension of the SPARQL syntax that caters for a fourth element in triple patterns and, for convenience, we will use the notation $\mu = \{ x_1/t_1, \ldots, x_n/t_n \}$ to indicate that $\mu(x_i) = t_i$, i.e. variable x_i is assigned to term t_i.

Example 6.9 (AnQL `optional`)**.** Suppose we are looking for Nightwish members during some time period and optionally the instrument they played. This query can be posed as follows:

```
SELECT $p $l $i WHERE {
   dbpedia:Nightwish foaf:member $p $l .
   OPTIONAL { $p mo:instrument $i $l } }
```

Take our example dataset from Data 6.1 extended with the following triples that indicate the instrument:

$$(\texttt{dbpedia:Marco\_Hietala}, \texttt{mo:instrument}, \texttt{:bass}) : [2005, 2009]$$

we will get the following answers:

$$\begin{aligned} \mu_1 &= \{ \$p/\texttt{dbpedia:Tarja\_Turunen}, \$l/[1996, 2005] \} \\ \mu_2 &= \{ \$p/\texttt{dbpedia:Marco\_Hietala}, \$l/[2001, 2012] \} \\ \mu_3 &= \{ \$p/\texttt{dbpedia:Marco\_Hietala}, \$l/[2005, 2009], \$i/\texttt{:bass} \} \ . \end{aligned}$$

The first two answers (μ_1 and μ_2) correspond to the answers in which the `optional` pattern is not satisfied, so we get the annotation values of $[1996, 2005]$ and $[2001, 2012]$, respectively, corresponding to the time that Tarja and Marco were members of Nightwish. In the third answer, the `optional` pattern is also matched and, in this case, the annotation value is restricted to the time when Marco is a member of Nightwish and we have information regarding the instrument he played.

Note that – as we will see – this first query will return as a binding for the annotation variable $\$l$ the periods where an instrument was played. A different query can be written that returns the periods of time an artist was a member of a band.

Example 6.10 (AnQL `optional` with `filter`)**.** The following query returns the Nightwish members during some time period that optionally played an instrument at some point during this time:

```
SELECT $p $l $i WHERE {
   dbpedia:Nightwish foaf:member $p $l .
   OPTIONAL { $p mo:instrument $i $l2 .
              FILTER ($l2 ⪯ $l) } }
```

Using the input data from Example 6.9, we obtain the following answers:

$$\begin{aligned}\mu_1 &= \{\, \$p/\texttt{dbpedia:Tarja\_Turunen}, \$l/[1996, 2005] \,\}\\ \mu_2 &= \{\, \$p/\texttt{dbpedia:Marco\_Hietala}, \$l/[2001, 2012] \,\}\\ \mu_3 &= \{\, \$p/\texttt{dbpedia:Marco\_Hietala}, \$l/[2001, 2012], \$i/\texttt{:bass} \,\} \quad .\end{aligned}$$

In this example the `filter` behaves as in SPARQL by removing from the answer set the mappings that do not make the `filter` expression true.

This query also exposes the issue of unsafe filters, noted in Angles and Gutiérrez (2008b) and we presented the semantics to deal with this issue in Definition 3.12.

6.2.2. Semantics

We can now define the semantics of AnQL queries by extending the notion of SPARQL BGP matching. Just as matching BGPs against RDF graphs is at the core of SPARQL semantics, matching BAPs against annotated RDF graphs is the heart of the evaluation semantics of AnQL.

We extend the notion of *substitution* to include a substitution of annotation variables in which we do not allow any assignment of an annotation variable to $\perp$ (of the domain D). An annotation value of $\perp$, although it is a valid answer for any triple, does not provide any additional information and thus is of minor interest. Furthermore this would contribute to increasing the number of answers unnecessarily.

Definition 6.7 (BAP evaluation)**.** *Let P be a BAP and G an annotated RDF graph. We define* evaluation $[\![P]\!]_G$ *as the list of substitutions that are* solutions *of P, i.e. $[\![P]\!]_G = \{\, \mu \mid G \models \mu(P) \,\}$, where $G \models \mu(P)$ means that any annotated triple in $\mu(P)$ is entailed by G.*

We can define the notion of solutions for BAP as the equivalent notion of answer sets for annotated conjunctive queries. As for SPARQL, we have:

Proposition 6.3. *Given an annotated graph G and a BAP P, the solutions of P are the same as the answers of the annotated query $q(vars(P)) \leftarrow P$ (where $vars(P)$ is the vector of variables in P), i.e. $ans(G, q) = [\![P]\!]_G$.*

For the extension of the SPARQL relational algebra to the annotated case we introduce – inspired by the definitions in (Pérez et al., 2009) – definitions of compatibility and union of substitutions:

Definition 6.8 ($\otimes$-compatibility)**.** *Two substitutions μ_1 and μ_2 are* $\otimes$-compatible *if and only if:*

(i) μ_1 and μ_2 are compatible for all the non-annotation variables, i.e. $\mu_1(x) = \mu_2(x)$ for any non-annotation variable $x \in dom(\mu_1) \cap dom(\mu_2)$; and
(ii) $\mu_1(\lambda) \otimes \mu_2(\lambda) \neq \perp$ for any annotation variable $\lambda \in dom(\mu_1) \cap dom(\mu_2)$.

This definition of *compatible solutions* is the same for non-annotated variables. For the case of shared annotation variables, consider as an example the temporal domain and two solutions μ_1 and μ_2 that share an annotation variable x. For μ_1 and μ_2 to be considered compatible the value their values for x must overlap:

- if $\mu_1(x) = [2001, 2005]$ and $\mu_2(x) = [2003, 2009]$, then $[2001, 2005] \otimes [2003, 2009] = [2003, 2005]$ and μ_1 and μ_2 will be considered compatible;
- on the other hand, if $\mu_1(x) = [2001, 2003]$ and $\mu_2(x) = [2005, 2009]$ then μ_1 and μ_2 will are not compatible.

Definition 6.9 ($\otimes$-union of substitutions)**.** *Given two $\otimes$-compatible substitutions μ_1 and μ_2, the* $\otimes$-union *of μ_1 and μ_2, denoted $\mu_1 \otimes \mu_2$, is as $\mu_1 \cup \mu_2$, with the exception that any annotation variable $\lambda \in dom(\mu_1) \cap dom(\mu_2)$ is mapped to $\mu_1(\lambda) \otimes \mu_2(\lambda)$.*

We now present the notion of evaluation for generic AnQL graph patterns. This consists of an extension of Definition 3.11:

Definition 6.10 (Evaluation, extends (Pérez et al., 2009, Definition 2))**.** *Let P be a BAP, P_1, P_2 annotated graph patterns, G an annotated graph and R a filter expression, then the evaluation $[\![\cdot]\!]_G$, is recursively defined as:*

- $[\![P]\!]_G = \{\, \mu \mid dom(\mu) = vars(P) \text{ and } G \models \mu(P) \,\}$
- $[\![P_1 \texttt{ and } P_2]\!]_G = \{\, \mu_1 \otimes \mu_2 \mid \mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G, \mu_1 \text{ and } \mu_2 \otimes\text{-compatible} \,\}$
- $[\![P_1 \texttt{ union } P_2]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
- $[\![P_1 \texttt{ filter } R]\!]_G = \{\mu \mid \mu \in [\![P_1]\!]_G \text{ and } R\mu \text{ is true}\}$
- $[\![P_1 \texttt{ optional } P_2[R]]\!]_G = \{\mu \mid \mu$ *meets one of the following conditions:*
  1. $\mu = \mu_1 \otimes \mu_2$ *if* $\mu_1 \in [\![P_1]\!]_G, \mu_2 \in [\![P_2]\!]_G, \mu_1$ *and* $\mu_2 \otimes$*-compatible, and* $R\mu$ *is true*
  2. $\mu = \mu_1 \in [\![P_1]\!]_G$ *and* $\forall \mu_2 \in [\![P_2]\!]_G$ *such that* μ_1 *and* $\mu_2 \otimes$*-compatible,* $R(\mu_1 \otimes \mu_2)$ *is true, and for all annotation variables* $\lambda \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_2(\lambda) \prec \mu_1(\lambda)$
  3. $\mu = \mu_1 \in [\![P_1]\!]_G$ *and* $\forall \mu_2 \in [\![P_2]\!]_G$ *such that* μ_1 *and* $\mu_2 \otimes$*-compatible,* $R(\mu_1 \otimes \mu_2)$ *is false* $\}$

Let R be a `filter` *expression and $x, y \in \mathbf{A} \cup L$, in addition to the* `filter` *expressions presented in Definition 3.11 we further allow the expressions presented next. The valuation of R on a substitution μ, denoted $R\mu$, is true if:*[7]
(9) $R = (x \preceq y)$ with $x, y \in dom(\mu) \cup L \wedge\ \mu(x) \preceq \mu(y)$;
(10) $R = p(\bar{\mathbf{z}})$ with $p(\bar{\mathbf{z}})\,\mu =$ true if and only if $p(\mu(\bar{\mathbf{z}})) =$ true, where p is a built-in predicate.
Otherwise $R\mu$ is false.

Please note that the cases for the evaluation of `optional` are compliant with the SPARQL specification (Prud'hommeaux and Seaborne, 2008), covering the notion of unsafe `filters` as presented by Angles and Gutiérrez (2008b). However, there are some peculiarities inherent to the annotated case. More specifically case 2.) introduces the side effect that annotation variables that are compatible between the mappings may have different values in the answer depending if the `optional` is matched or not. This is the behaviour demonstrated in Example 6.9.

In the `filter` expressions above, a built-in predicate p is any n-ary predicate p, where p's arguments may be variables (annotation and non-annotation ones), domain values of D, or values from **UL**; p has a fixed interpretation and we assume that the evaluation of the predicate can be decided in finite time.

Annotation domains may define their own built-in predicates that range over annotation values as in the following query:

Example 6.11 (AnQL query)**.** Consider our example dataset from Data 6.1 and that we want to know which band `dbpedia:Marco_Hietala` was a member of before 2005. This query can be expressed in the following way:

```
SELECT $band WHERE {
    $band foaf:member dbpedia:Marco_Hietala $l .
    FILTER(before($l, [2005])) }
```

For practical convenience, we retain in $[\![\cdot]\!]_G$ only "domain maximal answers". That is, let us define $\mu' \prec \mu$ if and only if:

(i) $\mu' \neq \mu$;

[7] We consider a simple evaluation of filter expressions where the "error" result is ignored, see Prud'hommeaux and Seaborne (2008, Section 11.3) for details.

(ii) $dom(\mu') = dom(\mu)$;
(iii) $\mu'(x) = \mu(x)$ for any non-annotation variable x; and
(iv) $\mu'(\lambda) \preceq \mu(\lambda)$ for all annotation variable λ.

Then, for any $\mu \in [\![P]\!]_G$ we remove any $\mu' \in [\![P]\!]_G$ such that $\mu' \prec \mu$.

6.2.3. Further Extensions of AnQL

In this section we will present extensions of Definition 6.10 to include features from the SPARQL 1.1 specification, such as variable assignments, aggregates, and solution modifiers.

Definition 6.11 (Assignment in AnQL)**.** *Let P be an annotated graph pattern and G an annotated graph, the evaluation of an* `ASSIGN` *statement is defined as:*

$$[\![P \ \texttt{ASSIGN}\ f(\bar{\mathbf{z}})\ AS\ z]\!]_G = \{\ \mu \mid \mu_1 \in [\![P]\!]_G, \mu = \mu_1[z/f(\mu_1(\bar{\mathbf{z}}))]\ \}$$

where

$$\mu[z/t] = \begin{cases} \mu \cup \{z/t\} & \text{if } z \notin dom(\mu) \\ (\mu \setminus \{z/t'\}) \cup \{z/t\} & \text{otherwise .} \end{cases}$$

Essentially, we assign to the variable z the value $f(\mu_1(\bar{\mathbf{z}}))$, which is the evaluation of the function $f(\bar{\mathbf{z}})$ with respect to a substitution $\mu_1 \in [\![P]\!]_G$.

Example 6.12 (Assignment in AnQL)**.** Using a built-in function we can retrieve for each artist the length of membership for any band:

```
SELECT $x $y $z WHERE {
    $y foaf:member $x $l .
    ASSIGN length($l) AS $z }
```

Here, the *length* built-in predicate returns, given a set of temporal intervals, the overall total length of the intervals.

We also introduce the `ORDERBY` clause where the evaluation of a $[\![P \ \texttt{ORDERBY}\ \$x]\!]_G$ statement is defined as the ordering of the solutions – for any $\mu \in [\![P]\!]_G$ – according to the values of $\mu(\$x)$. Ordering for non-annotation variables follows the rules in Prud'hommeaux and Seaborne (2008, Section 9.1). Similar to ordering in the query answering setting, we require that the set of values over which x ranges can be ordered. We can further extend the evaluation of AnQL queries with aggregate functions

$$@ \in \{\texttt{SUM}, \texttt{AVG}, \texttt{MAX}, \texttt{MIN}, \texttt{COUNT}, \oplus, \otimes\}$$

as follows:

Definition 6.12 (Grouping in AnQL)**.** *The evaluation of a* `GROUPBY` *statement is defined as:*[8]

$$[\![P \ \texttt{GROUPBY}(\bar{\mathbf{w}})\ \bar{@}\bar{\mathbf{f}}(\bar{\mathbf{z}})\ AS\ \bar{\alpha}]\!]_G = \{\ \mu \mid \mu'\ in\ [\![P]\!]_G, \mu = \mu'|_{\bar{\mathbf{w}}}[\alpha_i/@_i f_i(\mu'(\bar{\mathbf{z}}_i))]\ \}_{\mathsf{DISTINCT}}$$

where the variables $\alpha_i \notin var(P)$, $\bar{\mathbf{z}}_i \in var(P)$ and none of the `GROUPBY` *variables $\bar{\mathbf{w}}$ are included in the aggregation function variables $\bar{\mathbf{z}}_i$. Here, we denote by $\mu|_{\bar{\mathbf{w}}}$ the restriction of variables in μ to variables in $\bar{\mathbf{w}}$. Using this notation, we can also straightforwardly introduce projection, i.e. sub-SELECTs as an algebraic operator in the language covering another new feature of SPARQL 1.1:*

$$[\![\texttt{SELECT}\ \bar{\mathbf{V}}\ \{P\}]\!]_G = \{\ \mu \mid \mu'\ in\ [\![P]\!]_G, \mu = \mu'|_{\bar{\mathbf{v}}}\ \}\quad .$$

[8] In the expression, $\bar{@}\bar{\mathbf{f}}(\bar{\mathbf{z}})\ AS\ \bar{\alpha}$ is a concise representation of n aggregations of the form $@_i f_i(\bar{\mathbf{z}}_i)\ AS\ \alpha_i$ and $\{\ldots\}_{\mathsf{DISTINCT}}$ represents a duplicate removal operation.

Please note that the aggregator functions have a domain of definition and thus can only be applied to values of their respective domain. For example, `SUM` and `AVG` can only be used on numeric values, while `MAX`, `MIN` are applicable to any total order. The `COUNT` aggregator can be used for any finite set of values. The last two aggregation functions, namely $\oplus$ and $\otimes$, are defined by the annotation domain and thus can be used on any annotation variable.

Example 6.13 (Grouping in AnQL)**.** Suppose we want to know, for each artist, the average length of their membership with different bands. Then such a query will be expressed as:

```
SELECT $x $avgL WHERE {
    $y foaf:member $x $l .
    GROUPBY($x)
    AVG(length($l)) AS $avgL }
```

Essentially, we group by the artists, compute for each artist the time he was a member of a band (by means of the built-in function *length*), and compute the average value for each group. That is, if $g = \{\langle t, t_1\rangle, \ldots, \langle t, t_n\rangle\}$ is a group of tuples with the same value t for artist x, and value t_i for y, where each length of membership for t_i is l_i (computed as $length(\cdot)$), then the value of $avgL$ for the group g is $(\sum_i l_i)/n$.

6.3. AnQL Issues and Pitfalls

In this section we discuss some practical issues related to (i) the use of filters (Section 6.3.1); (ii) union of annotation values in the query (Section 6.3.2); and (iii) the representation of the temporal domain (Section 6.3.3).

6.3.1. Constraints vs Filters

Please note that `filters` do not act as *constraints* over the query. Consider the following example:

Example 6.14 (Constraints in AnQL)**.** Given the data from our dataset example and the following query:

```
SELECT $l1 $l2 WHERE {
    dbpedia:Nightwish foaf:member $p $l1 .
    dbpedia:Nightwish foaf:member dbpedia:Marco_Hietala $l2 . }
```

with an additional *constraint* that requires $\$l1$ to be "before" $\$l2$, we could expect the answer

$$\{\$l1/[1996, 2005], \$l2/[2006, 2012]\} \ .$$

This answer matches the following triples of our dataset:

$$(\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, \texttt{dbpedia:Marco\_Hietala}) : [2001, 2012]$$
$$(\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, \texttt{dbpedia:Tarja\_Turunen}) : [1996, 2005]$$

and satisfies the proposed *constraint*.

However, we require maximality of the annotation values in the answers, which in general do not exist in presence of *constraints*. For this reason, we do not allow general *constraints*.

6.3.2. Union of Annotations

The SPARQL `union` operator may also introduce some discussion when considering shared annotations between graph patterns.

Example 6.15 (Union of temporal annotations)**.** Take for example the following query and our dataset from Data 6.1 as input.

```
SELECT $l WHERE {
    { dbpedia:Nightwish foaf:member dbpedia:Marco_Hietala $l . }
    UNION
    { dbpedia:Tarot_(band) foaf:member dbpedia:Marco_Hietala $l . } }
```

Considering the temporal domain, the intuitive meaning of the query is "retrieve all time periods when Marco Hietala was a member of Nightwish or Tarot". In the case of `union` patterns the two instances of the variable $\$l$ are treated as two different variables. If the intended query would rather require treating both instances of the variable $\$l$ as the same, for instance to retrieve the time periods when Marco was a member of either Nightwish or Tarot but assuming we may not have information for one of the patterns, the query should rather look like:

```
SELECT $l WHERE {
    { dbpedia:Nightwish foaf:member dbpedia:Marco_Hietala $l1 . }
    UNION
    { dbpedia:Tarot_(band) foaf:member dbpedia:Marco_Hietala $l2 . }
    ASSIGN $l1 ∨ $l2 as $l }
```

where $\vee$ represents the domain specific built-in predicate for union of annotations.

6.3.3. Temporal Issues

Let us highlight some specific issues inherent to the temporal domain. Considering queries using Allen's temporal relations (Allen, 1983) (before, after, overlaps, etc.) as allowed by Tappolet and A. Bernstein (2009), we can pose queries like "find persons who were members of Nightwish before Troy". This query raises some ambiguity when considering that persons may have been members of the same band at different time intervals.

Example 6.16 (τSPARQL query)**.** Consider our dataset triples from Data 6.1 extended with the following triple:

$$(\texttt{dbpedia:Nightwish}, \texttt{foaf:member}, \texttt{dbpedia:Troy\_Donockley}) : \{[1996, 1999], [2006, 2008]\}$$

Tappolet and A. Bernstein (2009) consider this triple as two triples with disjoint intervals as annotations. For the following query in their language τSPARQL:

```
SELECT ?p WHERE {
    [?s1,?e1] dbpedia:Nightwish foaf:member ?p .
    [?s2,?e2] dbpedia:Nightwish foaf:member dbpedia:Troy_Donockley .
    [?s1,?e1] time:intervalBefore [?s2,?e2] }
```

we would get `dbpedia:Tarja_Turunen` as an answer although Troy was also a member of Nightwish when Tarja started. This is one possible interpretation of "before" over a set of intervals. In AnQL we could add different domain specific built-in predicates, representing different interpretations of "before". For instance, we could define binary built-ins:

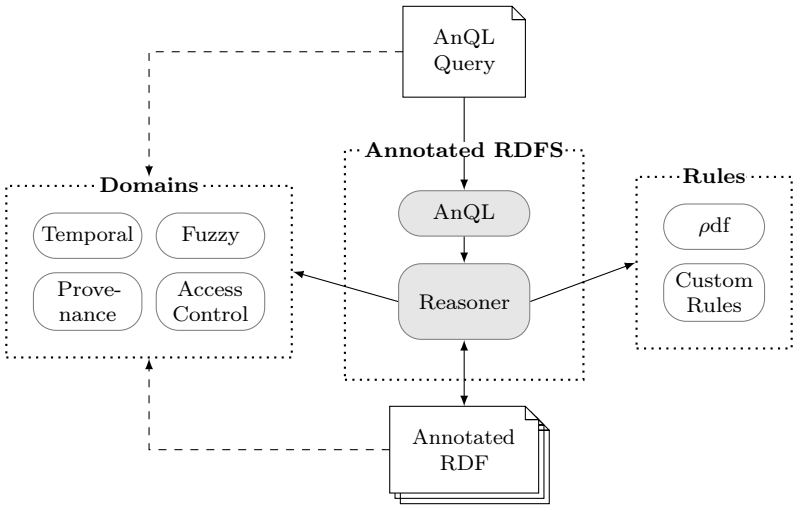


Figure 6.1.: Annotated RDFS implementation architecture

(i) beforeAny($A1, $A2), which is true if there exists *any* interval in annotation $A1 before an interval in $A2; or

(ii) respectively, a different built-in beforeAll($A1, $A2), which is only true if *all* intervals in annotation $A1 are before any interval in $A2.

Using the latter, an AnQL query would look as follows:

```
SELECT $p WHERE {
    dbpedia:Nightwish foaf:member $p $l1 .
    dbpedia:Nightwish foaf:member dbpedia:Tarja_Turunen $l2 .
    FILTER(beforeAll($l1,$l2)) }
```

This latter query gives no result, which might comply with people's understanding of "before" in some cases, while we also have the choice to adopt the behaviour of Tappolet and A. Bernstein (2009) by use of beforeAny instead.

6.4. Implementation Notes

Our prototype implementation is split into two distinct modules: one that implements the Annotated RDFS inferencing and the second module is an implementation of the AnQL query language that relies on the first module to retrieve the data. Our prototype implementation is based on SWI-Prolog's Semantic Web library (Wielemaker et al., 2008) and we present the architecture of the implementation in Figure 6.1.

For the syntax of the annotated RDF dataset we do not rely on any special serialisation but instead reuse other existing proposals e.g. using reification (Gutiérrez, Hurtado and Vaisman, 2007) or N-Quads (Cyganiak et al., 2009). Our engine parses the input RDF datasets into an internal representation, where each triple is represented using the `rdf/4` predicate, the arguments represent the *subject*, *predicate*, *object*, and *annotation value* of the triple.

The "Reasoner" module consists of a bottom-up engine that calculates the closure of a given "Annotated RDF" graph (or dataset). The variable components correspond to the specification of the given annotation domain ("Domains"); and the ruleset describing the inference rules and the way the annotation values should be propagated ("Rules"). The annotation domains are specified by the appropriate semi-ring operations and describe the default annotations for non-annotated triples.

Rules are specified using a high-level language to specify domain independent rules that abstracts away peculiarities of the underlying representation syntax:

Example 6.17 (RDFS subclass implementation rule)**.** In our implementation, the following rule provides *subclass inference* in the RDFS ruleset:

```
rdf(O, rdf:type, C2, V) <==
         rdf(O, rdf:type, C1, V1),
         rdf(C1, rdfs:subClassOf, C2, V2),
         infimum(V1, V2, V).
```

The Rules and Domains are independent of each other: it is possible to combine arbitrary rulesets and domains (see above).

The AnQL module also implemented in Prolog relies on the SPARQL implementation provided by the ClioPatria Semantic Web Server.[9] For the AnQL implementation, the domain specification needs to be extended with the grammar rules to parse an annotation value and any built-in functions specific to the domain.

Implementation of Specific Domains

For example, for the fuzzy domain the default value is considered to be 1 and the $\otimes$ and $\oplus$ operations are the *min* and *max* operations, respectively. The AnQL grammar rules consist simply of calling the parser predicate that parses a decimal value.

As for the temporal domain, we represent triple annotations as *ordered* lists of disjoint time intervals. This implies some additional care in the construction of the $\otimes$ and $\oplus$ operations. For the representation of $-\infty$ and $+\infty$ we use the `inf` and `sup` Prolog atoms, respectively. Concrete time points are represented as integers and we use a standard constraint solver over finite domains (CLPFD) in the $\otimes$ and $\oplus$ operations. The default value for non-annotated triples is `[inf,sup]`. The $\otimes$ operation is implemented as the recursive intersection of all elements of the annotation values, i.e. temporal intervals. The $\oplus$ operation is handled by constructing CLPFD expressions that evaluate the union of all temporal intervals. Again, the AnQL grammar rules take care of adapting the parser to the specific domain and we have defined the domain built-in operations described in Section 6.3.3.

6.5. Related Work

Adding annotations to logical statements was already proposed in the logic programming realm by Kifer and Subrahmanian (1992) who took a similar approach, where atomic formulas are annotated with a value taken from a lattice of annotation values, an annotation variable or a complex annotation, i.e. a function applied to annotation values or variables. Similarly, we can relate our work to annotated relational databases, especially Green et al. (2007) who provides a similar framework for relational algebra. After presenting a generic structure for annotations, they focus more specifically on the provenance domain.

Annotated RDF was first presented by Udrea et al. (2006); Udrea et al. (2010), where the authors define triples annotated with values taken from a *finite partial order*. In their work, triples are of the form $(s, p{:}\lambda, o)$, where the property, rather than the triple is annotated and furthermore represent RDF as a set of nodes and edges rather than extending the model theoretic semantics followed by the W3C. In our work, we rely on a richer, not necessarily finite, structure and provide additional inference capabilities when compared to Udrea et al. (2010), such as a more involved propagation of annotation values through

[9] `http://www.swi-prolog.org/web/ClioPatria/`, retrieved on 2012/04/10.

schema triples. Essentially, Udrea et al. do not provide an operation to combine annotations in RDFS inferences. The query language presented by Udrea et al. (2010) consists of conjunctive queries and, while SPARQL's BGPs are compared to their conjunctive queries, they do not consider extending SPARQL with the possibility of querying annotations. Furthermore, `optional`, `union` and `filter` SPARQL queries are not considered, which results in a subset of SPARQL that can be directly translated into their previously presented conjunctive query system.

In our initial approach the structure for representing annotations was defined as a residuated lattice (Straccia et al., 2010; Lopes, Polleres et al., 2010), which was later extended to the more general semiring structure by Buneman and Kostylev (2010). Furthermore, Buneman and Kostylev (2010) also show that once the RDFS inferences of an RDF graph have been computed for a specific domain, it is possible to reuse these inferences if the graph is annotated with a different domain. Based on this result, the authors define a universal domain, which is possible to transform to other domains by applying the corresponding transformations.

For the Semantic Web, several extensions of RDF were proposed in order to deal with *specific* domains such as truth of imprecise information (Mazzieri and Dragoni, 2008; Mazzieri and Dragoni, 2005; Mazzieri, 2004; Straccia, 2009; Lv et al., 2008), time (Gutiérrez, Hurtado and Vaisman, 2007; Pugliese et al., 2008; Tappolet and A. Bernstein, 2009), trust (Hartig, 2009; Schenk, 2008) and provenance (Dividino et al., 2009). These approaches are detailed in the following paragraphs.

Straccia (2009) presents Fuzzy RDF in a general setting where triples are annotated with a degree of truth in $[0, 1]$. For instance, "Rome is a big city to degree 0.8" can be represented with $(\texttt{Rome}, \texttt{type}, \texttt{BigCity}) : 0.8$; the annotation domain is $[0, 1]$. For the query language, it formalises conjunctive queries. Other similar approaches for Fuzzy RDF (Mazzieri and Dragoni, 2008; Mazzieri and Dragoni, 2005; Mazzieri, 2004) provide the syntax and semantics, along with RDF and RDFS interpretations of the annotated triples. Mazzieri (2004) describes an implementation strategy that relies on translating the Fuzzy triples into plain RDF triples by using reification. However these works focus mostly on the representation format and the query answering problem is not addressed.

Gutiérrez, Hurtado and Vaisman (2007) present the definitions of Temporal RDF, including reduction of the semantics of Temporal RDF graphs to RDF graphs and a sound and complete inference system. They show that entailment of Temporal graphs does not yield extra complexity beyond RDF entailment. Our Annotated RDFS framework encompasses this work by defining the temporal domain. The authors present conjunctive queries with built-in predicates as the query language for Temporal RDF, although they do not consider full SPARQL. Gutiérrez, Hurtado and Vaisman (2007) describe some further features such as a "Now" time point (which is a defined time point in the domain) and anonymous time points, allowing to state that a triple is true at some point. Adding anonymous time points would require us to extend the semi-ring by appropriate operators, e.g. $[2004, T] \oplus [T, 2008] = [2004, 2008]$ (where T is an anonymous time point). Pugliese et al. (2008) presents an optimised indexing schema for Temporal RDF, along with the notion of normalised Temporal RDF graph, and a query language for these graphs based on SPARQL. The indexing scheme consists of clustering the RDF data based on their temporal distance, for which several metrics are given. For the query language they only define conjunctive queries, thus ignoring some of the more advanced features of SPARQL. Tappolet and A. Bernstein (2009) present another approach to the implementation of Temporal RDF, where each temporal interval is represented as a named graph (Carroll, Bizer et al., 2005) containing all triples valid in that time period. Information about temporal intervals, such as their relative relations, start and end points, is asserted in the default graph. The τ-SPARQL query language allows to query the temporal RDF representation using an extended SPARQL syntax that can match the graph pattern against the snapshot of a temporal graph at any given time point and allows to query the start and endpoints of a temporal interval, whose values can then be used in other parts of the query.

SPARQL extensions towards querying trust have been presented by Hartig (2009), introducing a trust aware query language, tSPARQL, that includes a new constructor to access the trust value of a graph pattern. This value can then be used in other statements such as `FILTERs` or `ORDER`. Although focusing on trust, the approach is close to our general framework, introducing concepts similar to the ones presented in this chapter. However, a general framework was not presented. Also in the setting of trust management, Schenk (2008) defines a *bilattice* structure to model *trust* relying on the dimensions of knowledge and truth. The defined knowledge about trust in information sources can then be used to compute the trust of an inferred statement. An extension towards OWL is presented but there is no query language defined. Finally, this approach is used to resolve inconsistencies in ontologies arising from connecting multiple data sources.

Regarding provenance, in Delbru et al. (2008), the authors do not formalise the semantics and properties of the aggregation operation (simply denoted by $\wedge$) nor the exact rules that should be applied to correctly reason with provenance. Query answering is not tackled either. Flouris et al. (2009) provides more insight into the formalisation and details the rules by reusing (tacitly) Muñoz et al. (2007). They also provide a formalisation of a simple query language. However, the semantics they define is based on a strong restriction of ρdf (which is already a restriction of RDFS). As an example, they define the answers to the query $(\$x, \mathsf{type}, \$y, \$c)$ as the tuples (X, Y, C) such that there is a triple (X, type, Y, C) which can be inferred from only the application of rules (3a) and (3b) from the deductive system presented in Section 2.4.3. This means that a domain or range assertion would not provide additional answers to that type of query. Provenance also relates to the Named Graphs formalism (Carroll, Bizer et al., 2005) where one can identify distinct graphs with a URI. The name can be seen as an atomic provenance annotation. However, Named Graphs do not provide operations to combine the provenances. Yet, the formalism could be used as a possible syntactic solution for representing annotated triples.

Dividino et al. (2009) also present a generic extension of RDF to represent *meta information*, mostly focused on provenance and uncertainty. Such meta information is stored using named graphs and their extended semantics of RDF, denoted RDF$^+$, assumes a predefined vocabulary to be interpreted as meta information. However they do not provide an extension of the RDFS inference rules or any operations for combining meta information. The authors also provide an extension of the SPARQL query language, considering an additional expression that enables querying the RDF meta information.

Bonatti, Hogan et al. (2011) provide a framework for a specific combination of annotations (authoritativeness, rank, blacklisting, and provenance) within RDFS and (a variant of) OWL 2 RL. This work is orthogonal to ours, in that it does not focus on aspects of query answering, or providing a generic framework for combinations of annotations, but rather on scalable and efficient algorithms for materialising inferences for the specific combined annotations under consideration.

Different extensions of RDF and SPARQL focused on modelling spatial and temporal data were presented, namely stRDF (Koubarakis and Kyzirakos, 2010) and SPARQL-ST (Perry et al., 2011). SPARQL-ST focuses on extending the SPARQL query language relying on previous proposals such as Temporal RDF (Gutiérrez, Hurtado and Vaisman, 2007) and proposes a modelling of two dimensional geometries to represent the spatial coordinates in plain RDF. The extension of SPARQL is done by defining spatial and temporal variables and graph patterns and new filters and built-in conditions that operate over the temporal and spatial variables. Possible spatial filters allow to determine wether specific relations (e.g. equal, contains) hold between different geometries or to determine the distance between the geometries. Filtering the temporal variables is based on the Allen interval relations (Allen, 1983). stRDF and stSPARQL focus especially on representing sensor data, introducing triple annotations capable of representing moving trajectories of sensors and geometric areas where the sensors are deployed. Spatial data is represented by allowing RDF objects to be of a custom representation for geometries, whereas the temporal data is represented as an annotation over RDF triples. The stSPARQL query language

consists of an extension of SPARQL to consider the fourth element to query the temporal annotations, while spatial querying is based on filter expressions.

6.6. Conclusion

In this chapter we have presented a generalised RDF annotation framework that conservatively extends the RDFS semantics, along with an extension of the SPARQL query language to query annotated data. The framework presented here is generic enough to cover other proposals for RDF annotations and their query languages. Our approach extends the classical case of RDFS reasoning with features of different annotation domains, such as temporality, fuzzyness, or provenance. Furthermore, we presented a semantics for an extension of the SPARQL query language, AnQL, that enables querying RDF with annotations.

In the proposed data integration setting, this RDF extension can be used as a target data model, allowing to represent meta-information about the integrated data and thus allowing to resolve conflicts arising from the data integration process. In the next chapter we present a complete use case scenario where the defined language and data model are used to integrate data from different enterprise sources that may be protected by access control information. We also introduce the access control annotation domain that allows us to represent such annotated data and to enable sharing and querying only restricted sets of triples, on a per-user basis.

Part III.

An Integrated Use case

7. A Secure RDF Data Integration Framework

In this chapter we go back to the use case presented in Chapter 1, where we briefly mentioned the several software applications that enterprises use to manage their business: interactions with clients in a Customer Relationship Management (CRM) application, employee information in a Human Resources (HR) application, project documentation and company policies in a Document Management System (DMS) and records of time spent working on projects in a Timesheet System (TS). Heterogeneity of the data formats from the different software applications is not the only problem. In fact, as much of the information within the enterprise is highly sensitive, its integration could result in information leakage to unauthorised individuals.

In this chapter we build on the languages presented in the previous chapters to automatically extract data and access control information from the underlying databases and represent them as Annotated RDF graphs, providing a holistic view of data across the enterprise. This approach introduces a mechanism to enforce access control policies on the RDF graph along with a flexible and automatic way to represent and propagate the original access control policies.

In this chapter we define an annotation domain that models access control permissions as Annotated RDFS, specify the high-level system architecture required to enforce access control by relying on SPARQL, and illustrate how domain specific rules can be used to manage the access control annotations. First we present some common access control related terms that we use in this chapter:

Resources denote the information to be protected;
Users represent individuals requesting access to resources;
Groups are collections of users with common features (e.g. contributors, supervisors, and management);
Roles are commonly used to assign access rights to a set of individuals and groups, for example by department (e.g. human resources, sales and marketing) or task (e.g. insurance claim processing, reporting and invoicing).

7.1. The Access Control Annotation Domain

In this section we formalise our access control annotation domain, following the definitions presented in Section 6.1.2. We start by defining the entities and annotation values and then present the $\otimes$ and $\oplus$ domain operations. Finally, we briefly describe the implementation of the presented annotation domain.

7.1.1. Entities and Annotations

For the modelling of the access control domain consider, in addition to the previously presented sets of URIs $\mathbf{U}$, blank nodes $\mathbf{B}$, and literals $\mathbf{L}$, a set of credential elements $\mathbf{C}$. The elements of $\mathbf{C}$ are used to represent *users*, *roles*, and *groups*. To cater for *attribute based access control*, we consider a set `at` of pairs of form $k = v$, to be considered as attribute-value pairs, where $k, v \in \mathbf{L}$. For example "$age = 30$" or "$institute = DERI$" are elements of $\mathbf{T}$. We allow shortcuts to represent intervals of integers, for example

"$age = [25, 30]$" to indicate that all entities with attribute *age* between 25 and 30 are allowed access to the triple.

Considering an element $e \in \mathbf{CT}$, e and $\neg e$ are *access control elements*, where e is called a *positive* element and $\neg e$ is called a *negative* element.[1] An *access control statement* S consists of a set of access control elements and we further consider that S is in *consistent* iff for any element $e \in \mathbf{CT}$, only one among e and $\neg e$ may appear in S. This restriction avoids *conflicts*, where a statement is attempting to both *grant* and *deny* access to a triple. Furthermore, we can define a partial order between statements as $S_1 \leqslant S_2$ iff $S_1 \subseteq S_2$ that can be used to eliminate *redundant* access permissions: if a user is granted access by statement S_2, he will also be granted access by statement S_1 (and thus S_2 can be removed). Finally, an *Access Control List (ACL)* consists of a set of access control statements and an ACL is considered *consistent* iff each statement it contains is consistent and not redundant. In our domain representation, only consistent ACLs are considered as annotation values. Intuitively, an annotation value specifies which entities have read permission to the triple, or are denied access when the annotation is preceded by $\neg$.

Example 7.1 (Access Control List)**.** We are considering the following set of entities $\mathbf{C} = \{jb, js, st, it\}$, where jb and js are employee usernames and st and it are shorthand for *softwareTester* and *informationTechnology*, respectively. The following annotated triple:

$$\tau \colon [[it], [st, \neg js]]$$

states that the entities identified with it or st (except if the js credential is also present) have read access to the triple τ.

An ACL A can be considered as a non-recursive Datalog with negation (nr-datalog$^\neg$) program, where access control statement $s \in A$ corresponds to the body of a rule in the Datalog program. The head of the Datalog rules is a reserved literal $access \notin \mathbf{CT}$ and the evaluation of the Datalog program determines the access permission to a triple for a specific set of credentials.

The set of user credentials is assumed to be provided by an external authentication service and consists of elements of $\mathbf{CT}$, which equate to a non-empty ACL representing the entities associated with the user. We further assume that this ACL consists of only one positive statement, i.e. the ACL will contain only one statement with all the entities associated with the user and does not contain any negative elements.

Example 7.2 (Datalog Representation of an ACL)**.** Consider the annotation example presented in Example 7.1. The nr-datalog$^\neg$ program corresponding to the ACL $[[it], [st, \neg js]]$ is:

$$access \leftarrow it.$$
$$access \leftarrow st, \neg js.$$

The set of credentials of the user *session*, provided by the external authentication system eg. $[[jb, it]]$, are considered the facts in the nr-datalog$^\neg$ program.

Further domain specific information, for example hierarchies between the access control entities, can be encoded as extra rules within the nr-datalog$^\neg$ program. These extra rules can be used to provide *implicit* credentials to a user, allowing the access control to be specified based on credentials that the authentication system does not necessarily assign to a user.

[1] Here we are using $\neg e$ to represent strong negation. In our access control domain representation, $\neg e$ indicates that e will be specifically *denied* access.

Example 7.3 (Credential Hierarchies)**.** Considering that the entity *emp* represents all the employees within a specific company, and that jb and js correspond to employee usernames (as presented in Example 7.1), the following rules can be added to the nr-datalog$^\neg$ program from Example 7.2:

$$emp \leftarrow js.$$
$$emp \leftarrow jb.$$

These rules ensure that both jb and js are given access when the credential *emp* is required in an annotation value.

7.1.2. Annotation Domain

We now turn to the annotation domain operations $\otimes$ and $\oplus$ that, as presented in Section 6.1, allow for the combination of annotation values when performing RDFS inference. A naive implementation of these domain operations may produce ACLs that are not consistent (and would not be considered valid annotation values). To avoid such invalid ACLs, we rely on a normalisation step that ensures the result is a valid annotation value by checking for redundant statements and applying a conflict resolution policy (described below) if necessary.

Definition 7.1 (Normalise)**.** *Let A be an ACL. We define the reduction of A into its consistent form, denoted $norm(A)$, as:*

$$norm(A) = \{\ normalise(s_i) \mid s_i \in A \text{ and } \nexists s_j \in A, i \neq j \text{ such that } s_i \leqslant s_j\ \}$$

where the normalisation of a statement s, denoted $normalise(s)$ consists of applying the conflict resolution policy described below.

We say that an access statement contains a *conflict* if it contains a positive and negative access control element of the same entity, e.g. $[jb, \neg jb]$. There are different ways to resolve conflicts in the annotation statements: apply a (i) *brave conflict resolution* (allow access); or (ii) *safe conflict resolution* (deny access). This is achieved during the normalisation step, represented by the *normalise* function, by removing the appropriate element: $\neg jb$ for brave or jb for safe conflict resolution. In our current modelling, we are assuming safe conflict resolution.

The $\oplus$ operation for the access control domain consists of the union of the annotations and then performing the normalisation operation. The intuitive behaviour is that of creating a new nr-datalog$^\neg$ program that consists of the union of the rules of the programs of both original annotations. Formally,

$$A_1 \oplus_{ac} A_2 = norm(A_1 \cup A_2)\ \ .$$

In turn, the $\otimes$ operation consists of merging the rules belonging to both annotation programs and then performing the normalisation and conflict resolution. This corresponds to further restricting the statements from both annotations to only those entities that are provided access by both annotations. Formally, the $\otimes$ operations corresponds to:

$$A_1 \otimes_{ac} A_2 = norm(\{\ s_1 \cup s_2 \mid s_1 \in A_1 \text{ and } s_2 \in A_2\ \})\ \ ,$$

where $s_1 \cup s_2$ represents the set theoretical union. Unlike the $\oplus_{ac}$ operation, the $\otimes_{ac}$ may produce conflicts in the annotation statements.

Example 7.4 (Domain Operations)**.** Consider the annotations $A_1 = [[jb], [js], [\neg it]]$ and $A_2 = [[it]]$. The $\otimes$ operation is used when inferring new triples, and thus the resulting annotation should provide

access to the resulting triple only to entities that are allowed to access all the premisses:

$$A_1 \otimes_{ac} A_2 = [[jb, it], [js, it], [\neg it]] \ .$$

Please note that the aforementioned conflict resolution mechanism has simplified $[\neg it, it]$ into $[\neg it]$. On the other hand the $\oplus$ operation is used to combine annotations when the same triple is deduced from different inference steps. Thus, combining annotations with the $\oplus$ operations should result in providing access to all the entities with are allowed to access the premises:

$$A_1 \oplus_{ac} A_2 = [[jb], [js], [\neg it], [it]] \ .$$

Lastly, the smallest and largest annotation value in the access control domain $\bot_{ac}$ and $\top_{ac}$, respectively correspond to an empty nr-datalog$^\neg$ program and another that provides access to all entities $e \in \mathbf{CT}$: $\bot_{ac} = []$ and $\top_{ac} = \{\, [e], [\neg e] \mid e \in \mathbf{CT} \,\}$. The $\bot_{ac}$ annotation value element indicates that the annotated triple is not accessible to any entity, since no annotation statements will provide access to the triple, and an annotation value of $\top_{ac}$ states that the triple is considered *public*, since any credential contained in the user session will provide access to the triple.

Definition 7.2 (Access Control Annotation Domain)**.** *Let* $\mathbf{F}$ *be the set of annotation values over* $\mathbf{CT}$*, i.e. consistent ACLs. The access control annotation domain is formally defined as:*

$$D_{ac} = \langle \mathbf{F}, \oplus_{ac}, \otimes_{ac}, \bot_{ac}, \top_{ac} \rangle \ .$$

For our access control domain model, the $\bot_{ac}$ is considered the *default* annotation for any non-annotated triple, which implicitly denies access to the triple.

This modelling of the access control domain can be extended to consider other permissions, like *update*, and *delete* simply by extending the annotation to an n-tuple of propositional formulæ[2] $\langle P, Q, \ldots \rangle$, where P specifies the formula for *read* permission, Q for *update* permission, etc. This extension allows to use the defined domain operations simply extended to operate over the corresponding components of the tuple. A *create* permission has a different behaviour as it would not be attached to any specific triple but rather as a graph-wide permission and thus is not considered in this modelling. In this chapter, we are focusing only on *read* permissions in the description of the domain and thus restrict the modelling to a single propositional formula. It is worth noting that the support for *create* and *update* of RDF is only included in the forthcoming W3C SPARQL 1.1 Recommendation (Harris and Seaborne, 2012).

7.1.3. Domain Implementation

According to the prototype described in Section 6.4, the implementation of the access control annotation domain consists of a Prolog module that is imported by the reasoner. This module defines the domain operations $\otimes_{ac}$ and $\oplus_{ac}$, represented as the predicates `infimum/3` and `supremum/3`, respectively. The annotation values are represented simply by using *lists*, in this case lists of lists, following the definitions presented in the previous section.

The implementation of the $\oplus_{ac}$ operation involves concatenating the list representation of both annotations and then performing the normalisation operation. As for the $\otimes_{ac}$ operation, we follow a similar procedure to the $\oplus_{ac}$ operation, with the additional step of applying one of the previously presented *brave* and *safe* conflict resolution methods. The evaluation of the nr-datalog$^\neg$ program can be performed based on the representation of the annotation values, by checking if the list of credentials of a user is a superset of any of the positive literals of the statements of our annotation values and also that it does not contain any of the negative literals of the statement.

[2] One formula, two formulæ.

```
@prefix : <http://urq.deri.org/enterprise#> .

:westportCars rdf:type :Company "[[jb]]".
:westportCars :netIncome 1000000 .
:joeBloggs :worksFor :westportCars .
:joeBloggs :salary 80000 "[[jb]]".
:johnSmith :worksFor :westportCars .
:johnSmith :salary 40000 "[[js]].
```

Data 7.1: Access Control Annotated RDFS

An example of RDF data annotated with Access Control information, where the salary information is only available to the respective employee, is presented in Data 7.1. In this figure we are representing the RDF triples and annotation element using the N-Quads RDF serialisation (Cyganiak et al., 2009). Using AnQL, the extension of the SPARQL query language described in Section 6.2, it is possible to perform queries that take into consideration the access control annotations. An example of an AnQL query over this data is presented in the following example:

Example 7.5 (AnQL Query Example)**.** This query specifies that we are interested in the salary of employees that someone with the permissions $[[jb, st, it]]$ is allowed to access.

```
SELECT * WHERE { ?p :salary ?s "[[jb, st, it]]" }
```

The answers for this query (when matched against Data 7.1) under SPARQL semantics, i.e. if the annotation would be omitted, would be:

$$\{\, \{\, \$p \rightarrow \texttt{:joeBloggs}, \$s \rightarrow 80000 \,\}, \{\, \$p \rightarrow \texttt{:johnSmith}, \$s \rightarrow 40000 \,\} \,\} \quad .$$

However, with the inclusion of domain annotations, an AnQL query engine must also perform the following check: $[[jb, st, it]]$ satisfies the nr-datalog$^{\neg}$ program λ, where λ is the program represented by the annotation of each matched triple, thus yielding only the following answer:

$$\{\, \{\, \$p \rightarrow \texttt{:joeBloggs}, \$s \rightarrow 80000 \,\} \,\} \quad .$$

7.2. An Access Control Aware Data Integration Architecture

This section describes the minimal set of components necessary for a data integration and access control enforcement framework. It provides an overview of our implementation of each component (based on the languages and models described in this thesis) and presents an experimental evaluation of our prototype, which focuses on: (i) the *RDB2RDF* data integration; (ii) the *reasoning engine*; and (iii) the *query engine*. The aim of this evaluation is simply to show the feasibility of our approach and, although we present different dataset sizes, at this point we are not looking at improving scalability and thus do not propose any kind of optimisations.

We start by presenting a combination of the XSPARQL and AnQL languages, that allows us to query the heterogeneous sources and create the target Annotated RDF graph.

7.2.1. Combining XSPARQL and AnQL

Next we present the combination of the XSPARQL language, as presented in Chapter 4, with the AnQL query language described in Chapter 6. This combination caters for the creation and querying of

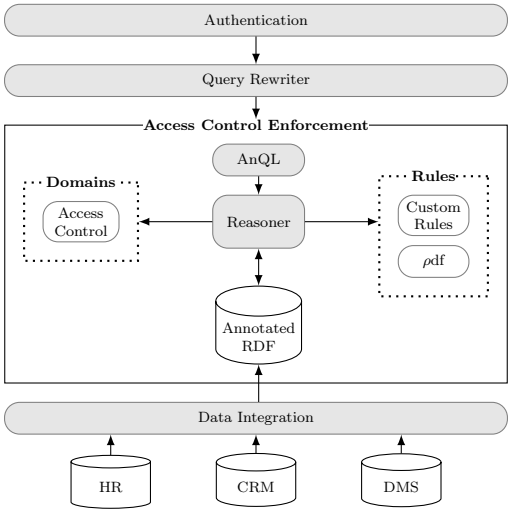


Figure 7.1.: RDF Data Integration and Access Control Enforcement Framework

Annotated RDF graphs using the XSPARQL language. For the purposes of this thesis, namely the data integration use case, we are mostly interested in creating the Annotated RDF data.

In XSPARQL we extend the syntax of *SparqlForClause*s and *ConstructClause*s to cater for the fourth element (as presented in Chapter 6), thus allowing us to create and query the Annotated RDF graphs, respectively. Considering this extended expression syntax, the semantics of *SparqlForClause*s, presented in Section 4.2, can be changed to follow the AnQL semantics instead of the SPARQL semantics. Conversely, in the XSPARQL implementation (described in Section 5.1) we can replace the ARQ SPARQL engine with our own AnQL prototype implementation (cf. Section 6.4).

For the creation of RDF graphs, the current implementation of the XSPARQL language (described in Section 5.1.2) relies on creating a string representation of the RDF graph in Turtle notation. For the creation of Annotated RDF graphs, we similarly extend this string representation to cater for an Annotated RDF graph according to the N-Quads representation (Cyganiak et al., 2009).

Following the N-Quads specification, we represent the annotation value as an RDF literal, which also allows us to implement the extension of XSPARQL independently of the annotation domain. We also introduce a new generic type, which we call `AnnotationLiteral`, which will be the type of any annotation values and variables. In XSPARQL we follow the restriction that annotation variables and non-annotation variables should be distinct in the query and this newly introduced type ensures that we can enforce this restriction in nested XSPARQL queries. Similar to AnQL, we assume the sharing of variables is possible only by using domain specific functions that handle the appropriate type conversions.

The combination of XSPARQL and Annotated RDFS also introduces inferencing capabilities into XSPARQL by reusing the annotated inference rules presented in Section 6.1.5. Notably the classical domain (cf. Section 6.1.4) caters for the classical RDFS inferences. A proper formalisation of this combination is beyond the scope of this thesis, however a possible starting point is the SPARQL 1.1 Entailment Regimes specification (Glimm and Ogbuji, 2012), which introduces other entailment regimes (beyond simple RDF entailment) into the upcoming SPARQL 1.1 specification.

7.2.2. Access Control Enforcement Framework

An overview of the proposed framework is depicted in Figure 7.1, which is composed of two main modules: *Data Integration* and *Access Control Enforcement*. The Data Integration module is responsible for the conversion of existing relational data and access control policies to RDF. Whereas the Access Control

Enforcement module caters for the management of access rights and enables authenticated users to query their RDF data. Noticeably, one component we do not tackle in this chapter is the *authentication* component, which can be achieved by relying on WebId (Sporny et al., 2011) and self-signed certificates. The enforcement of the access control is performed by relying on the *query rewriter* component, that expands a provided SPARQL query with the credentials of the authenticated user.

Data Integration

The Data Integration module is responsible for the extraction of data and associated access rights from the underlying relational databases. The information extracted is subsequently transformed into Annotated RDF using the combination of XSPARQL and AnQL described in Section 7.2.1. Ideally, the data integration step would be carried out in conjunction with a domain expert, for example to assist in defining an R2RML (Das, Sundara et al., 2012) mapping or XSPARQL query that extracts and converts the relational data into RDF. This chapter focusses primarily on retrieving data from relational databases as the enterprise systems we worked with stored their data in relational format.[3]

Example 7.6 (XSPARQL+AnQL)**.** The sample query below demonstrates how information about a project can be extracted from an enterprise timesheet system.

```
@prefix : <http://urq.deri.org/enterprise#>

for p.Proj_Code, p.Proj_Desc, rp.Res_Code
from Projects p, ResPrj rp
where p.Proj_Code = rp.Proj_Code
construct {
       :{$p.Proj_Code} a :Project {fn:concat("[[",$rp.Res_Code,"]]")};
       :Client {$p.Cust_Code} {fn:concat("[[",$rp.Res_Code,"]]")} }
```

The query consists of a *SQLForClause* clause that extracts the data from the two underlying relations and the *ConstructClause* in turn is used to generate N-Quads from the results of the database query.

Access Control Enforcement

This component is based on our implementation of the Annotated RDFS framework (presented in Figure 6.1) where the annotation domain is fixed to access control. The integrated data retrieved from the original relational databases is stored as Annotated RDF.

Reasoner. For this component we consider two distinct forms of inference: (a) data inference, where new triples are deduced from existing ones (such as the RDFS rules); and (b) access rights inference, where new permissions are deduced from existing ones. In our prototype, the reasoning component is implemented by the extension of the RDFS inference rules presented in Section 6.1.3.

In many LOB applications, two forms of hierarchies are considered: (i) hierarchies between entities in the access control annotations; and (ii) hierarchies between common resources in the data. Hierarchies of form (i) were considered in Section 7.1.1 by adding rules to the nr-datalog$^{\neg}$ program that evaluates the annotations. As for (ii), permissions granted to a resource should inherited by all of the resources children. Such inheritance chain can be broken by explicitly specifying permissions at a lower level in the tree.

Considering our access control domain modelling and the use-case of extracting data (and permissions) from their original sources, one option is to incorporate this business logic into the extraction process. In this case, the extraction query must have information on how to propagate the access permissions and apply them to all the necessary triples. Another option is to use domain specific rules, which our reasoner

[3]The data and queries presented in this chapter were developed and executed in collaboration with a DERI industry partner. Any data presented here was anonymised.

is capable of processing, in order to propagate the access permissions or to ensure any domain specific policies. Such rules can be written in a similar way to the Annotated RDFS rules, described in Section 6.4, giving us access to the existing data and annotations and allowing us to create new Annotated RDF triples or update existing ones.

Example 7.7 (Domain Specific Rule)**.** Consider, in an enterprise scenario, that an existing policy states that if an employee is given access to a `Company` record, as per the following triple (C, `type`, `:Company`), that employee should be given access to all triples regarding that company. Such a policy can be enforced by using the following rule:

$$\frac{(C, \texttt{type}, \texttt{:Company}) \colon \lambda_1,\ (C, P, O) \colon \lambda_2}{(C, P, O) \colon \lambda_1 \oplus_{ac} \lambda_2} \ ,$$

where C, P, O and λ_1, λ_2 are variables. Applying this rule to the sample dataset presented in Figure 7.1, would cause the access permission of the triple $(\texttt{:westportCars}, \texttt{type}, \texttt{:Company}) \colon [[jb]]$ to be propagated to the second triple, yielding the following new annotated triple:

$$(\texttt{:westportCars}, \texttt{:netIncome}, \texttt{1000000}) \colon [[jb]] \ .$$

Query Rewriter

It is possible to use AnQL directly to query RDF data annotated with access control information, as presented in Example 7.5. However, allowing the end user to perform AnQL queries is not secure since one could bypass the access control due to the lack of enforcement of the supplied credentials.

Our proposed solution for the enforcement of the access control is based on query rewriting. The user is allowed to write SPARQL queries and the system transparently extends each triple pattern of the provided query with the user credentials as annotation value, thus generating an AnQL query. This generated AnQL query is then executed against the Annotated RDF graph, which guarantees that the user can only access the triples based on the credentials provided. This query rewriting step relies on information provided by the external authentication system: a user session represents information regarding an authenticated user in the system and contains, among others, the user credentials. The user credentials should be represented as an annotation control element and thus can be easily added into any SPARQL BGP to obtain an AnQL BAP.

7.2.3. Experimental Evaluation

The benchmark system is a virtual machine, running a 64-bit edition of Windows Server 2008 R2 Enterprise, located on a shared server. The virtual machine has an Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, with 4 shared processing cores and 5GB of dedicated memory. For the evaluation we extract both the data and the access rights from two separate software application databases using XSPARQL. The different datasets (DS_1, DS_2, DS_3, and DS_4) use the same databases, tables, and XSPARQL queries and differ only on the number of records that are retrieved from the databases. Table 7.2 provides a summary of each dataset, stating the number of database records queried, the number of triples generated, and the size of the N-Quads representation of the triples. Furthermore, Table 7.2 includes the run time of the data extraction process and the run time of importing the data into our Prolog implementation. Figure 7.2 provides a high level overview of the times for each of the datasets.

Based on this simple experiment we have hints that the extraction process and the loading of triples into Prolog behave linearly but more data intensive tests are still required. As the inferencing times are highly dependent on both the rules and the data further experimentation is required in this area.

As for the evaluation of the AnQL engine we used the following queries, denoted Q_1, Q_2, and Q_3:

Table 7.1.: Access Control dataset description

	DS_1	DS_2	DS_3	DS_4
database records	8854	16934	33095	65417
triples	44775	88300	175345	349430
file size (MB)	6.1	12.1	23.7	47.6

Table 7.2.: Access Control dataset generation and load times

	DS_1	DS_2	DS_3	DS_4
RDB2RDF (sec)	26	42	82	153
Import (sec)	2.69	4.74	9.17	18.94

Q_1**:** we retrieved all data

```
SELECT * WHERE { ?s ?p ?o ?λ1 }
```

Q_2**:** we queried the data for a specific user

```
SELECT * WHERE { ?s ?p ?o "[[jb]]" }
```

Q_3**:** we queried the data for a specific role

```
SELECT * WHERE { ?s ?p ?o "[[:administrators]]" }
```

The evaluation results of these three queries over the different datasets is presented in Table 7.3 and depicted in Figure 7.2. These results calculated as an average of 3 response times and show an overhead for the evaluation of annotations Q_2 and Q_3.

7.3. Related Work

The topic of access control has been long studied in relational databases and the approach of enforcing the access policies by query rewriting was also considered for the Quel query language by Stonebraker and Wong (1974). However, the presented system does not rely on annotating the relational data but rather access control is specified using constraints over the user credentials, which are then included in the rewritten query. An overview of common issues, existing models and languages for access control is provided by di Vimercati et al. (2005).

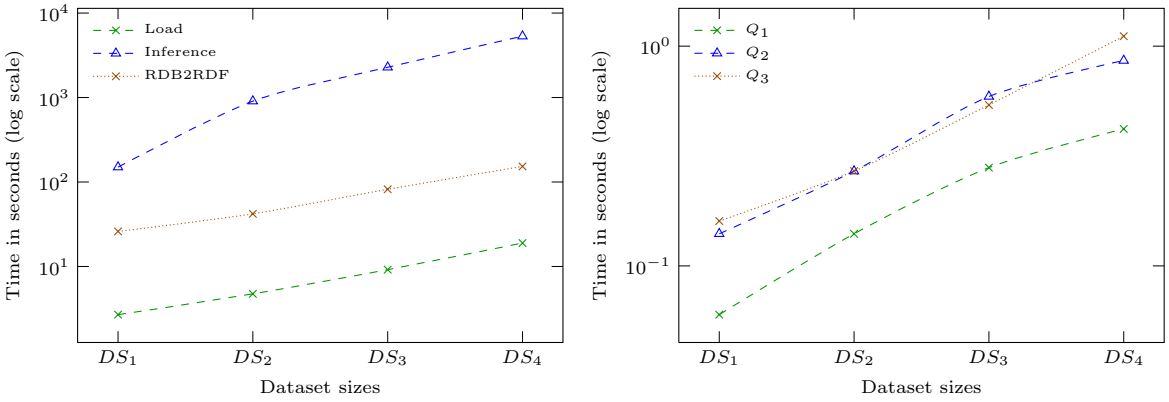
For the Semantic Web, well known policy languages such as KAoS (Bradshaw et al., 1997), Rei (Kagal and Finin, 2003) and PROTUNE (Bonatti, De Coi et al., 2009). Although such languages enable policy specification using RDF and OWL, in their current form they do not support reasoning based on RDF data. These policy languages are complimentary to our work as they can be mapped to our annotations using rules.

Dietzold and Auer (2006) describe the requirements an RDF store needs from a Semantic Wiki perspective. Apart from the necessary requirements on efficiency and scalability, the authors refer the need for access control on a triple level and the need to integrate the structure of the organisation in the access control methods. The described system relies on a query engine (SPARQL is mentioned but no details are given) and a rule processor to decide the access control enforcement at query time. The system we propose in this chapter caters for both of these requirements and also integrates the access control into the annotation query language.

Hollenbach et al. (2009) present the possibility of maintaing metadata on the RDF data to enforce access control and discuss, as possible extensions of their model, some of the work presented here, e.g. using

Table 7.3.: Query execution time in seconds for the different Access Control datasets.

	DS_1	DS_2	DS_3	DS_4
Q_1	0.06	0.14	0.28	0.42
Q_2	0.14	0.27	0.59	0.86
Q_3	0.16	0.27	0.54	1.11



(a) Dataset load times

(b) Query execution times

Figure 7.2.: Load and query execution times for the different Access Control datasets

rules for specifying access control. Providing access control on a resource level is also left as an open question, one we are tackling by the specification of rules. The extension of SPARQL is not considered.

Similar access control annotations are considered attached to axioms in an ontology by Knechtel and Stuckenschmidt (2010) and Baader et al. (2009) and are used to allow access to subsets of the ontology to specific users and also apply such annotations to the problem of determining the minimal set of axioms that are necessary to support a certain conclusion. Although the setting is different to the one presented in this chapter, some of the algorithms for efficient annotation calculation may be ported to our modelling.

Some work on extending query languages was presented by Abel et al. (2007), however this work pre-dates the SPARQL query language. In a similar fashion to the work proposed in this chapter, their policy enforcement is also done by a query rewriting step, however their query rewriting does not consists of including the user credentials but rather replicating the access policies within the query. They also consider access control policies that both grant and restrict access to data.

7.4. Conclusion

In this chapter we proposed an Access Control model, which can be used to protect RDF data and demonstrate how a combination of Annotated RDFS and SPARQL can be used to control access to integrated enterprise data. This model is based on the Annotated RDFS framework presented in Chapter 6 and attaches the access control information on a triple basis, i.e. each RDF triple can contain different annotation values. This solution provides a flexible representation method for the access control annotations, based on propositional formulæ to define which entities have access to the triple. However, when considering large number of triples, challenges arise with respect to optimal access control policy administration. To tackle this issue we propose permission management through specifying domain-specific inference rules for the annotation domain. We also suggest a possible implementation structure for a framework to enforce the access control based on rewriting a SPARQL query into an AnQL query.

Part IV.

Conclusion

8. Conclusions

In this thesis we presented a novel query language, called XSPARQL, that combines the SQL, XQuery, and SPARQL query languages in order to provide transformations between relational, XML, and RDF data. We also presented extensions of the RDF data model, called Annotated RDFS, and of the SPARQL query language, called AnQL, that cater for fine-grained meta-information, which we consider necessary to accurately represent integrated data. We included initial optimisation strategies for a particular category of XSPARQL queries, namely those containing nested `for` expressions that improved the evaluation times for transformations between the different data models.

The main hypothesis of this thesis, presented in Section 1.3, states that:

> **Efficient data integration over heterogeneous data sources can be achieved by: (i) a query language that allows to access data adhering to different formats in the original sources (without the need for data transformation); (ii) a set of optimisations that allow for efficient query evaluation in such a query language; and (iii) an interchange representation format with support for meta-information, allowing to represent temporal, uncertain, provenance, or even access-control information.**

The core chapters of this thesis present the components that validate this hypothesis:

Chapter 4 introduced the XSPARQL query language, which allows us to easily bridge the heterogeneous data sources and perform transformations between data adhering to different models. This language combines the syntax and semantics of different query languages: it is based on the syntax of the XQuery language and defines new expressions that access the heterogeneous data models. The result is an expressive language that enables writing arbitrary transformations between data adhering to the different models and thus can be used in several data integration scenarios. In this chapter, we have introduced several examples for such transformations and have also shown that XSPARQL can be used to implement the new W3C specification for converting relational data to RDF: RDB2RDF.

Chapter 5 describes our implementation of the XSPARQL language, along with an experimental evaluation of this language using a newly proposed benchmark suite. This evaluation has revealed the queries that incur a greater penalty for accessing the heterogeneous sources: nested `for` expressions in which the inner clause accesses an RDF source. For these cases we have proposed different optimisations, for which we also presented a benchmark evaluation with the obtained performance increases. The different optimisations rely on applying techniques from SQL or XQuery for nested expressions, such as performing query unnesting or, when possible, pushing the query into a single format.

Chapter 6 presents our proposed extension of the RDF data model where it is possible to annotate RDF triples with meta-information from a specific domain. The domains we defined in this chapter allow for attaching temporal information to a triple specifying time periods when the triple is considered valid, fuzzy information that specifies a degree to which the triple is considered valid, or provenance information that can be used to determine which data sources contributed to the generation of the

triple. We proposed a general extension that is able to encapsulate all of these domains and also extends the RDFS inference rules and SPARQL query language in a domain-independent fashion.

Although RDF is being increasingly used for representing integrated data, as we have argued in this thesis, RDF alone is not enough. The proposed extension of RDF caters for necessary dimensions of the integration process. Especially the presented Access Control domain, has not been tackled before to such granularity. As highlighted in Halevy, Ashish et al. (2005) this is a much needed feature:

> When retrieving information from diverse sources, ensuring security, e.g. ensuring that only authorised users get access to the information they seek, continues to be an underserved area.

The Linked Open Data community has so far focused on freely available data, emphasising the "Open" part. However, in order for RDF to become widely adopted in enterprise environments, it requires mechanisms to secure and protect data.

Chapter 7 presented an approach that is a stepping stone towards such a system: we defined a new annotation domain where RDF triples can be annotated with information regarding which entities are allowed to access it. In Chapter 7 we used XSPARQL to access the different underlying sources, transform the data into Annotated RDF with access control information, and introduced some possible AnQL queries over this annotated data. The presented framework also defines a rewriting step in which SPARQL queries can be automatically expanded into AnQL queries to provide secure access to the RDF data.

8.1. Critical Assessment

Even after this thesis, the problem of data integration is not yet solved! The presented XSPARQL transformation language enables existing data warehousing and mediator approaches to integrate information via a query and transformation language. However, no *one-size-fits-all* solution exists today nor is it likely to exist in the near future. As Halevy, Ashish et al. (2005) state:

> (...) the greatest cost in an ETL model is the human cost of setup and administration: understanding the query requirements, understanding the data sources, building and maintaining the complex processes that clean and integrate the data.

For an enterprise scenario, a proper analysis of the benefits and drawbacks of each approach needs to be carried out. One problem is that applications for data integration rapidly become outdated; e.g. as enterprise software applications evolve, the data integration applications need to be updated accordingly. Although a similar drawback is still present when the integration is performed via a query language, a clear evaluation semantics can improve the data integration task not only by enabling optimisations but also in the subsequent adaptation of the query to changes in the underlying data sources.

In our optimisations chapter (Chapter 5) we also asserted that we need different kinds of optimisations for different data models. We have observed this fact when we tried to apply the optimisations for SPARQL nested expressions to SQL nested expressions. It is possible that this is a simple implementation issue and that different implementations of the XSPARQL language would not present these results. Further investigation would be required to determine why these optimisations do not cary over across different data models. Most likely this discrepancy is due to the support structures in place in the database management system, ranging from the persistent storing of data to the indexing provided over the stored data. Such structures allow for the efficient evaluation of nested simple queries, as opposed to our optimised implementation that collects and joins the data in XQuery. For the nested queries over RDF data, each iteration incurs the increased cost of loading the dataset alongside the normal query evaluation.

8.2. Future Directions

Some possible future directions for the work presented in this thesis include improvements to the data lifting direction and the definition of a core declarative model for the XSPARQL language that caters for accessing the relational, XML, and (Annotated) RDF data. Another necessary, yet challenging future topic is to devise an update language over the different data models. Finally, a declarative description of data sources would allow an XSPARQL-based integration framework to be built. These topics are now briefly described.

Data Lifting

The roots of the XSPARQL language have come from the need to transform existing RDF data into (arbitrary) XML and, as such we have focused on the *lowering* direction. The wider community adoption of the XSPARQL language has also highlighted interest in the *lifting* direction (which is reflected in this thesis). For this features, several extensions can be made to the language with respect to the implementation, moving beyond our current representation for RDF graphs (based on strings) into a more integrated approach – for example, relying on representations for RDF graphs from existing RDF stores – would allow a more direct translation to be implemented, e.g. inserting the generated RDF graph directly into the store.

Regarding the language, new approaches for lifting can be devised, for instance, support for the construction of nested predicate-object pairs when the subject has already been determined. Furthermore, it remains to be determined if optimisations for the lifting process are necessary.

Declarative Model

In Chapter 5 we have shown that nested queries can be evaluated efficiently by applying different rewriting strategies for XSPARQL queries. However, all of these rewriting strategies were ad-hoc whereas the definition of a declarative algebra model would help to correctly and systematically study further optimisations for XSPARQL. This declarative model must include a representative subset of the XSPARQL language with known complexity bounds, while still allowing queries over heterogeneous sources to be performed.

Possible starting points for such a declarative model are in the work by Koch (2006), where some complexity results for a non-recursive core fragment of XQuery are presented. Another possible approach is to explore the long standing mapping from relational algebra to Datalog, where more recent work by Grust, Mayr et al. (2010) presents translations of XQuery to SQL. Relatedly, Polleres (2007); Angles and Gutiérrez (2008b) present translations from SPARQL to Relational Algebra. These works seem to indicate valid starting points for further research on equivalences and optimisations in our language.

Using the declarative model, it is also possible to check the equivalence between any proposed optimisations and also, in a similar approach to Levy et al. (1996), allow to assign a cost function to each source in order to be able to calculate (near) optimal query plans.

Update Language

Another important feature for a data integration language is the capability to perform updates over the original sources. This is also acknowledged by Halevy, Ashish et al. (2005):

> However, there's more to data than reads. What about updates? A virtual database update model is often not the best fit for enterprise integration scenarios.

The current XSPARQL language specification already allows to query data contained in relational, XML, and RDF datastores. However, updating data in these datastores is still not possible. We plan to extend

the XSPARQL language to a full data manipulation language allowing for the update, insert, and delete of data contained in RDF triplestores. Analogously to our combination of query languages, we will aim at combining common data manipulation languages for XML and RDF, such as SPARQL Update (Gearon et al., 2012) and XQuery Update (Robie et al., 2011).

However, such an update language over integrated data is not a trivial task, since updates over the integrated data need to be reflected in the original sources.

Query Language Abstraction

The proposed query language and future declarative model can form the basis for a more complex data integration system. One possible approach is to devise a declarative representation for data sources, along with rules that specify how to integrate their data. Based on this declarative abstraction, it is possible to provided automated mappings from the source descriptions and transformation rules into XSPARQL queries, thus implementing the data integration process in a straightforward fashion.

For the declarative description of the data sources we can attempt to leverage existing vocabularies and ontologies that describe existing data sources, for example providing an abstraction layer over existing sensor readings, relational databases, or social web feeds.

Bibliography

Abel, F., Coi, J. L. D., Henze, N., Koesling, A. W., Krause, D. and Olmedilla, D. (2007). Enabling Advanced and Context-Dependent Access Control in RDF Stores. In K. Aberer, K.-S. Choi, N. F. Noy, D. Allemang, K.-I. Lee, L. J. B. Nixon, ... P. Cudré-Mauroux (Eds.), *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007* (Vol. 4825, pp. 1–14). Springer.

Abiteboul, S. (1997). Querying Semi-Structured Data. In F. N. Afrati and P. G. Kolaitis (Eds.), *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings* (Vol. 1186, pp. 1–18). Springer.

Abiteboul, S., Benjelloun, O. and Milo, T. (2002). Web services and data integration. In T. W. Ling, U. Dayal, E. Bertino, W. K. Ng and A. Goh (Eds.), *3rd International Conference on Web Information Systems Engineering (WISE 2002), 12-14 December 2002, Singapore, Proceedings* (pp. 3–6). IEEE Computer Society.

Abiteboul, S., Buneman, P. and Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann.

Abiteboul, S., Hull, R. and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Adida, B. and Birbeck, M. (Eds.). (2008). *RDFa Primer – Bridging the Human and Data Webs*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/`

Afanasiev, L. and Marx, M. (2008). An Analysis of XQuery Benchmarks. *Information Systems*, *33*(2), 155–181.

Agrawal, P., Benjelloun, O., Sarma, A. D., Hayworth, C., Nabar, S. U., Sugihara, T. and Widom, J. (2006). Trio: A System for Data, Uncertainty, and Lineage. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, ... Y.-K. Kim (Eds.), *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006* (pp. 1151–1154). ACM.

Akhtar, W., Kopecky, J., Krennwallner, T. and Polleres, A. (2008). XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. In *Proceedings of the 5th European Semantic Web Conference (ESWC2008)* (pp. 432–447). Tenerife, Spain: Springer.

Allen, J. F. (1983). Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, *26*(11), 832–843.

Amagasa, T., Yoshikawa, M. and Uemura, S. (2000). A Data Model for Temporal XML Documents. In M. T. Ibrahim, J. Küng and N. Revell (Eds.), *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings* (Vol. 1873, pp. 334–344). Springer.

Angles, R. and Gutiérrez, C. (2008a). Survey of Graph Database Models. *ACM Computing Surveys*, *40*(1).

Angles, R. and Gutiérrez, C. (2008b). The Expressive Power of SPARQL. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin and K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings* (Vol. 5318, pp. 114–129). Springer.

Angles, R. and Gutiérrez, C. (2010). SQL Nested Queries in SPARQL. In A. H. F. Laender and L. V. S. Lakshmanan (Eds.), *Proceedings of the 4th Alberto Mendelzon International Workshop on Foundations of Data Management, Buenos Aires, Argentina, May 17-20, 2010* (Vol. 619). CEUR-WS.org.

Angles, R. and Gutiérrez, C. (2011). Subqueries in SPARQL. In P. Barceló and V. Tannen (Eds.), *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management, Santiago, Chile, May 9-12, 2011* (Vol. 749). CEUR-WS.org.

Arenas, M., Gutiérrez, C. and Pérez, J. (2009). On the Semantics of SPARQL. In R. D. Virgilio, F. Giunchiglia and L. Tanca (Eds.), (pp. 281–307). Springer.

Arenas, M., Kantere, V., Kementsietsidis, A., Kiringa, I., Miller, R. J. and Mylopoulos, J. (2003). The Hyperion Project: From Data Integration to Data Coordination. *SIGMOD Record*, *32*(3), 53–58.

Arenas, M., Prud'hommeaux, E. and Sequeda, J. (Eds.). (2012). *A Direct Mapping of Relational Data to RDF*. W3C Recommendation. Retrieved September 27, 2012, from `http://www.w3.org/TR/2012/REC-rdb-direct-mapping-20120927/`

Auer, S., Dietzold, S., Lehmann, J., Hellmann, S. and Aumueller, D. (2009). Triplify – Light-Weight Linked Data Publication from Relational Databases. In J. Quemada, G. León, Y. S. Maarek and W. Nejdl (Eds.), *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009* (pp. 621–630). ACM.

Baader, F., Knechtel, M. and Peñaloza, R. (2009). A Generic Approach for Large-Scale Ontological Reasoning in the Presence of Access Restrictions to the Ontology's Axioms. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta and K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference* (Vol. 5823, pp. 49–64). Lecture Notes in Computer Science. Springer.

Baru, C. K., Gupta, A., Ludäscher, B., Marciano, R., Papakonstantinou, Y., Velikhov, P. and Chu, V. (1999). XML-Based Information Mediation with MIX. In A. Delis, C. Faloutsos and S. Ghandeharizadeh (Eds.), *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA* (pp. 597–599). ACM Press.

Battle, S. (2006). Gloze: XML to RDF and back again. In *Proceedings of the First Jena User Conference*.

Beckett, D. (Ed.). (2004). *RDF/XML Syntax Specification (Revised)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`

Beckett, D. and Broekstra, J. (Eds.). (2008). *SPARQL Query Results XML Format*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/`

Beckett, D. (2010). RDF Syntaxes 2.0. In *Proceedings of the W3C Workshop on RDF Next Steps*. Palo Alto, CA, USA. Retrieved March 27, 2012, from `http://www.w3.org/2009/12/rdf-ws/papers/ws11`

Beckett, D. and Berners-Lee, T. (2011). Turtle - Terse RDF Triple Language. W3C Team Submission. Retrieved March 27, 2012, from `http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/`

Belhajjame, K., Deus, H., Garijo, D., Klyne, G., Missier, P., Soiland-Reyes, S. and Zednik, S. (2012). *PROV Model Primer*. W3C. Retrieved September 24, 2012, from `http://www.w3.org/TR/2012/WD-prov-primer-20120724/`

Benjelloun, O., Sarma, A. D., Halevy, A. Y., Theobald, M. and Widom, J. (2008). Databases with uncertainty and lineage. *VLDB Journal*, *17*(2), 243–264.

Berners-Lee, T. (2005). Notation 3 Logic. W3C Design Issues. Retrieved March 27, 2012, from `http://www.w3.org/DesignIssues/N3Logic`

Berners-Lee, T., Fielding, R. T. and Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. Retrieved March 27, 2012, from `http://tools.ietf.org/html/rfc3986`

Berners-Lee, T., Hendler, J. and Lassila, O. (2001). The Semantic Web. *Scientific American*, *284*(5), 34–43.

Bernstein, P. A. and Haas, L. M. (2008). Information Integration in the Enterprise. *Communications of the ACM*, *51*(9), 72–79.

Berrueta, D., Labra, J. E. and Herman, I. (2008). XSLT+SPARQL : Scripting the Semantic Web with SPARQL embedded into XSLT stylesheets. In C. Bizer, S. Auer, G. A. Grimmes and T. Heath (Eds.), *4th Workshop on Scripting for the Semantic Web*. Tenerife.

Bikakis, N., Gioldasis, N., Tsinaraki, C. and Christodoulakis, S. (2009). Querying XML Data with SPARQL. In S. S. Bhowmick, J. Küng and R. Wagner (Eds.), *Database and Expert Systems Applications, 20th International Conference, DEXA 2009, Linz, Austria, August 31 - September 4, 2009. Proceedings* (Vol. 5690, pp. 372–381). Springer.

Biron, P. V. and Malhotra, A. (Eds.). (2004). *XML Schema Part 2: Datatypes Second Edition*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`

Bischof, S., Decker, S., Krennwallner, T., Lopes, N. and Polleres, A. (2012). Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, *1*, 147–185.

Bizer, C. (2003). D2R MAP - A Database to RDF Mapping Language. In *World Wide Web Conference 2003 (Posters)*.

Bizer, C., Heath, T. and Berners-Lee, T. (2009). Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, *5*(3), 1–22.

Bizer, C., Lehmann, J., Kobilarov, G., Auer, S., Becker, C., Cyganiak, R. and Hellmann, S. (2009). DBpedia - A Crystallization Point for the Web of Data. *Journal of Web Semantics*, *7*(3), 154–165.

Bohring, H. and Auer, S. (2005). Mapping XML to OWL Ontologies. In K. P. Jantke, K.-P. Fähnrich and W. S. Wittig (Eds.), *Marktplatz Internet: von E-Learning bis E-Payment, 13. Leipziger Informatik-Tage, LIT 2005, 21.-23. September 2005, Leipzig* (Vol. 72, pp. 147–156). GI.

Bonatti, P. A., De Coi, J. L., Olmedilla, D. and Sauro, L. (2009). Rule-Based Policy Representations and Reasoning. In *Semantic techniques for the web* (Chap. 4, pp. 201–232).

Bonatti, P. A., Hogan, A., Polleres, A. and Sauro, L. (2011). Robust and scalable Linked Data reasoning incorporating provenance and trust annotations. *Journal of Web Semantics*, *9*(2), 165–201.

Bradshaw, J. M., Dutfield, S., Benoit, P. and Woolley, J. D. (1997). KAoS: Toward an industrial-strength open agent architecture. In *Software Agents* (pp. 375–418).

Bray, T., Hollander, D., Layman, A., Tobin, R. and Thompson, H. S. (Eds.). (2009). *Namespaces in XML 1.0 (Third Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2009/REC-xml-names-20091208/`

Bray, T., Paoli, J. and Sperberg-McQueen, C. M. (Eds.). (2010). *XML Path Language (XPath) 2.0 (Second Edition)*. World Wide Web consortium. Retrieved March 27, 2012, from `http://www.w3.org/TR/2010/REC-xpath20-20101214/`

Bray, T., Paoli, J., Sperberg-Mcqueen, C. M., Maler, E. and Yergeau, F. (Eds.). (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2008/REC-xml-20081126/`

Brickley, D. and Guha, R. V. (Eds.). (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`

Buneman, P. (1997). Semistructured Data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (pp. 117–121). PODS '97. New York, NY, USA: ACM.

Buneman, P. and Kostylev, E. (2010). Annotation Algebras for RDFS. In *The Second International Workshop on the role of Semantic Web in Provenance Management (SWPM-10)*. CEUR Workshop Proceedings.

Carroll, J. J., Bizer, C., Hayes, P. J. and Stickler, P. (2005). Named graphs. *Journal of Web Semantics*, *3*(4), 247–267.

Carroll, J. J. and Stickler, P. (2004). *TriX, RDF Triples in XML* (tech. rep. No. HPL-2003-268). HP Labs. Retrieved March 27, 2012, from `http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html`

Ceri, S. and Gottlob, G. (1985). Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, *11*(4), 324–345.

Chamberlin, D., Robie, J., Boag, S., Fernández, M. F., Siméon, J. and Florescu, D. (Eds.). (2010). *XQuery 1.0: An XML Query Language (Second Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2010/REC-xquery-20101214/`

Chinnici, R., Moreau, J.-J., Ryman, A. and Weerawarana, S. (Eds.). (2007). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C Recommendation. Retrieved March 27, 2012, from `http://www.w3.org/TR/2007/REC-wsdl20-20070626/`

Clark, K. G., Feigenbaum, L. and Torres, E. (Eds.). (2008). *SPARQL Protocol for RDF*. W3C Recommendation. Retrieved March 27, 2012, from `http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/`

Cluet, S., Delobel, C., Siméon, J. and Smaga, K. (1998). Your Mediators Need Data Conversion! In L. M. Haas and A. Tiwary (Eds.), *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA* (pp. 177–188). ACM Press.

Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, *13*(6), 377–387.

Codd, E. F. (1980). Data Models in Database Management. In M. L. Brodie and S. N. Zilles (Eds.), *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, June 23-26, 1980* (Vol. 11, *2*, pp. 112–114). ACM Press.

Connolly, D. (Ed.). (2007). *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*. W3C Recommendation. Retrieved March 27, 2012, from `http://www.w3.org/TR/2007/REC-grddl-20070911/`

Corby, O., Kefi-Khelif, L., Cherfi, H., Gandon, F. and Khelif, K. (2009). *Querying the Semantic Web of Data using SPARQL, RDF and XML* (tech. rep. No. 6847). Institut National de Recherche en Informatique et en Automatique.

Cowan, J. and Tobin, R. (Eds.). (2004). *XML Information Set (Second Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-xml-infoset-20040204/`

Cui, Y., Widom, J. and Wiener, J. L. (2000). Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems*, *25*(2), 179–227.

Cyganiak, R. (2005). *A relational algebra for SPARQL*. HP-Labs. Retrieved March 27, 2012, from `http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html`

Cyganiak, R., Harth, A. and Hogan, A. (2009). N-Quads: Enxtending N-Triples with Context. Retrieved March 27, 2012, from `http://sw.deri.org/2008/07/n-quads/`

Das, S. and Srinivasan, J. (2009). Database Technologies for RDF. In S. Tessaris, E. Franconi, T. Eiter, C. Gutiérrez, S. Handschuh, M.-C. Rousset and R. A. Schmidt (Eds.), *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009, Brixen-Bressanone, Italy, August 30 - September 4, 2009, Tutorial Lectures* (Vol. 5689, pp. 205–221). Springer.

Das, S., Sundara, S. and Cyganiak, R. (Eds.). (2012). *R2RML: RDB to RDF Mapping Language*. W3C Recommendation. Retrieved September 27, 2012, from `http://www.w3.org/TR/2012/REC-r2rml-20120927/`

Delbru, R., Polleres, A., Tummarello, G. and Decker, S. (2008). Context Dependent Reasoning for Semantic Documents in Sindice. In A. Fokoue, Y. Guo and J. H. T. Liebig (Eds.), *4th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2008)*.

Deursen, D. V., Poppe, C., Martens, G., Mannens, E. and Walle, R. V. d. (2008). XML to RDF Conversion: A Generic Approach. In *International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution, 2008 (AXMEDIS '08)* (pp. 138–144). Washington, DC, USA: IEEE Computer Society.

di Vimercati, S. D. C., Samarati, P. and Jajodia, S. (2005). Policies, Models, and Languages for Access Control. In S. Bhalla (Ed.), *Databases in Networked Information Systems, 4th International Workshop, DNIS 2005, Aizu-Wakamatsu, Japan, March 28-30, 2005, Proceedings* (Vol. 3433, pp. 225–237). Springer.

Dietzold, S. and Auer, S. (2006). Access Control on RDF Triple Stores from a Semantic Wiki Perspective. In C. Bizer, S. Auer and L. Miller (Eds.), *Proceedings of 2nd Workshop on Scripting for the Semantic Web at ESWC, Budva, Montenegro.* (Vol. 183).

Dillnut, R. (2006). Surviving the information explosion [knowledge management]. *Engineering Management Journal*, *16*(1), 39–41.

Dividino, R. Q., Sizov, S., Staab, S. and Schueler, B. (2009). Querying for Provenance, Trust, Uncertainty and other Meta Knowledge in RDF. *Journal of Web Semantics*, *7*(3), 204–219.

Doan, A. and Halevy, A. Y. (2005). Semantic Integration Research in the Database Community: A Brief Survey. *AI Magazine*, *26*(1), 83–94.

Draper, D., Fankhauser, P., Fernández, M., Malhotra, A., Rose, K., Rys, M., ... Wadler, P. (Eds.). (2010). *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/`

Draper, D., Halevy, A. Y. and Weld, D. S. (2001a). The Nimble Integration Engine. In *SIGMOD Conference* (pp. 567–568).

Draper, D., Halevy, A. Y. and Weld, D. S. (2001b). The Nimble XML Data Integration System. In D. Georgakopoulos and A. Buchmann (Eds.), *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany* (pp. 155–160). IEEE Computer Society.

Droop, M., Flarer, M., Groppe, J., Groppe, S., Linnemann, V., Pinggera, J., ... Zugal, S. (2008). Embedding XPath Queries into SPARQL Queries. In J. Cordeiro and J. Filipe (Eds.), *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008* (pp. 5–14).

Eisenberg, A. and Melton, J. (2001). SQL/XML and the SQLX Informal Group of Companies. *SIGMOD Record*, *30*(3), 105–108.

Eisenberg, A. and Melton, J. (2004). Advancements in SQL/XML. *SIGMOD Record*, *33*(3), 79–86.

Erling, O. and Mikhailov, I. (2007). RDF Support in the Virtuoso DBMS. In S. Auer, C. Bizer, C. Müller and A. V. Zhdanova (Eds.), *The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany* (Vol. 113, pp. 59–68). GI.

Fernández, M. F., Malhotra, A., Marsh, J., Nagy, M. and Walsh, N. (Eds.). (2010). *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`

Fischer, P., Florescu, D., Kaufmann, M. and Kossmann, D. (2011). Translating SPARQL and SQL to XQuery. In *XML Prague 2011* (pp. 81–98).

Flouris, G., Fundulaki, I., Pediaditis, P., Theoharis, Y. and Christophides, V. (2009). Coloring RDF Triples to Capture Provenance. In A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta and K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2009, 8th International Semantic Web Conference* (Vol. 5823, pp. 196–212). Lecture Notes in Computer Science. Springer.

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J. D., ... Widom, J. (1997). The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, *8*(2), 117–132.

Gearon, P., Passant, A. and Polleres, A. (Eds.). (2012). *SPARQL 1.1 Update*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2012/WD-sparql11-update-20120105/`

Glimm, B. and Ogbuji, C. (Eds.). (2012). *SPARQL 1.1 Entailment Regimes*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2012/WD-sparql11-entailment-20120105/`

Goh, C. H., Madnick, S. E. and Siegel, M. D. (1994). Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment. In *Proceedings of the third international conference on Information and knowledge management* (pp. 337–346). CIKM '94. New York, NY, USA: ACM.

Grant, J. and Beckett, D. (Eds.). (2004). *RDF Test Cases*. World Wide Web consortium. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-rdf-testcases-20040210/`

Gray, A. J. G., Gray, N. and Ounis, I. (2009). Can RDB2RDF Tools Feasibily Expose Large Science Archives for Data Integration? In L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, ... E. P. B. Simperl (Eds.), *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings* (Vol. 5554, pp. 491–505). Lecture Notes in Computer Science. Springer.

Green, T. J., Karvounarakis, G. and Tannen, V. (2007). Provenance Semirings. In L. Libkin (Ed.), *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China* (pp. 31–40). ACM Press.

Groppe, S., Groppe, J., Linnemann, V., Kukulenz, D., Hoeller, N. and Reinke, C. (2008). Embedding SPARQL into XQuery/XSLT. In R. L. Wainwright and H. Haddad (Eds.), *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008* (pp. 2271–2278). ACM.

Grust, T., Mayr, M. and Rittinger, J. (2010). Let SQL Drive the XQuery Workhorse (XQuery Join Graph Isolation). In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, ... F. Özcan (Eds.), *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings* (Vol. 426, pp. 147–158). ACM.

Grust, T., Rittinger, J. and Teubner, J. (2008). Pathfinder: XQuery Off the Relational Shelf. *IEEE Data Engineering Bulletin*, *31*(4), 7–14.

Grust, T., Sakr, S. and Teubner, J. (2004). XQuery on SQL Hosts. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakeley and K. B. Schiefer (Eds.), *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004* (pp. 252–263). Morgan Kaufmann.

Gutiérrez, C., Hurtado, C. A. and Mendelzon, A. O. (2004). Foundations of Semantic Web Databases. In *PODS* (pp. 95–106).

Gutiérrez, C., Hurtado, C. A. and Vaisman, A. A. (2007). Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, *19*(2), 207–218.

Halevy, A. Y., Ashish, N., Bitton, D., Carey, M. J., Draper, D., Pollock, J., ... Sikka, V. (2005). Enterprise Information Integration: Successes, Challenges and Controversies. In F. Özcan (Ed.), *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005* (pp. 778–787). ACM.

Halevy, A. Y., Rajaraman, A. and Ordille, J. J. (2006). Data Integration: The Teenage Years. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, ... Y.-K. Kim (Eds.), *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006* (pp. 9–16). ACM.

Harris, S. and Seaborne, A. (Eds.). (2012). *SPARQL 1.1 Query Language*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2012/WD-sparql11-query-20120105/`

Hartig, O. (2009). Querying Trust in RDF Data with tSPARQL. In L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, ... E. P. B. Simperl (Eds.), *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings* (Vol. 5554, pp. 5–20). Lecture Notes in Computer Science. Springer.

Hayes, J. and Gutiérrez, C. (2004). Bipartite Graphs as Intermediate Model for RDF. In S. A. McIlraith, D. Plexousakis and F. van Harmelen (Eds.), *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings* (Vol. 3298, pp. 47–61). Springer.

Hayes, P. (Ed.). (2004). *RDF Semantics*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F. and Rudolph, S. (Eds.). (2009). *OWL 2 Web Ontology Language Primer*. W3C. Retrieved September 18, 2012, from `http://www.w3.org/TR/2009/REC-owl2-primer-20091027/`

Hollenbach, J., Presbrey, J. and Berners-Lee, T. (2009). Using RDF Metadata To Enable Access Control on the Social Semantic Web. In T. Tudorache, G. Correndo, N. Noy, H. Alani and M. Greaves (Eds.), *Proceedings of the Workshop on Collaborative Construction, Management and Linking of Structured Knowledge (CK2009)* (Vol. 514). CEUR-WS.org.

ISO. (1986). *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Geneva, Switzerland: International Organization for Standardization.

Kagal, L. and Finin, T. (2003). A Policy Language for a Pervasive Computing Environment. In *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks* (pp. 63–74). IEEE Computer Society.

Katz, H., Chamberlin, D., Kay, M., Wadler, P. and Draper, D. (2003). *XQuery from the Experts: A Guide to the W3C XML Query Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Kay, M. (Ed.). (2007). *XSL Transformations (XSLT) Version 2.0*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2007/REC-xslt20-20070123/`

Kepser, S. (2004). A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages®*.

Kifer, M. and Subrahmanian, V. S. (1992). Theory of Generalized Annotated Logic Programming and its Applications. *Journal of Logic Programming*, *12*, 335–367.

Klement, E. P., Mesiar, R. and Pap, E. (2000). *Triangular Norms*. Trends in Logic - Studia Logica Library. Kluwer Academic Publishers.

Knechtel, M. and Stuckenschmidt, H. (2010). Query-Based Access Control for Ontologies. In P. Hitzler and T. Lukasiewicz (Eds.), *Web Reasoning and Rule Systems - Fourth International Conference,*

RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings (Vol. 6333, pp. 73–87). Springer.

Knublauch, H., Hendler, J. A. and Idehen, K. (2011). SPIN - Overview and Motivation. W3C Member Submission. Retrieved March 27, 2012, from `http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222/`

Koch, C. (2006). On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *ACM Transactions on Database Systems*, *31*(4), 1215–1256.

Koubarakis, M. and Kyzirakos, K. (2010). Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral and T. Tudorache (Eds.), *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part I* (Vol. 6088, pp. 425–439). Springer.

Krennwallner, T., Lopes, N. and Polleres, A. (2009). XSPARQL: Semantics. W3C member submission. Retrieved March 27, 2012, from `http://www.w3.org/Submission/2009/SUBM-xsparql-semantics-20090120/`

Levy, A. Y., Rajaraman, A. and Ordille, J. J. (1996). Querying Heterogeneous Information Sources Using Source Descriptions. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan and N. L. Sarda (Eds.), *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (pp. 251–262). Morgan Kaufmann.

Lloyd, J. W. (1987). *Foundations of Logic Programming, 2nd Edition*. Springer.

Lopes, N., Bischof, S., Decker, S. and Polleres, A. (2011). On the Semantics of Heterogeneous Querying of Relational, XML and RDF Data with XSPARQL. In P. Moura and V. B. Nogueira (Eds.), *Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA2011) – Computational Logic with Applications Track*. Lisbon, Portugal.

Lopes, N., Bischof, S., Erling, O., Polleres, A., Passant, A., Berrueta, D., ... Zaremba, M. (2010). RDF and XML: Towards a unified query layer. In *Proceedings of the W3C Workshop on RDF Next Steps*. Palo Alto, CA, USA.

Lopes, N., Kirrane, S., Zimmermann, A., Polleres, A. and Mileo, A. (2012). A Logic Programming approach for Access Control over RDF. In A. Dovier and V. S. Costa (Eds.), *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)* (Vol. 17, pp. 381–392). Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Lopes, N., Krennwallner, T., Polleres, A., Akhtar, W. and Stéphane Corlosquet. (2009). XSPARQL: Implementation and Test-cases. W3C Member Submission. Retrieved March 27, 2012, from `http://www.w3.org/Submission/2009/SUBM-xsparql-implementation-20090120/`

Lopes, N. and Polleres, A. (2011). Integrating RDF and XML with XSPARQL. 2011 Semantic Technology Conference. Retrieved March 27, 2012, from `http://semtech2011.semanticweb.com/sessionPop.cfm?confid=62&proposalid=3897`

Lopes, N., Polleres, A., Straccia, U. and Zimmermann, A. (2010). AnQL: SPARQLing Up Annotated RDFS. In P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, ... B. Glimm (Eds.), *International Semantic Web Conference (1)* (Vol. 6496, pp. 518–533). Lecture Notes in Computer Science. Springer.

Lopes, N., Zimmermann, A., Hogan, A., Lukácsy, G., Polleres, A., Straccia, U. and Decker, S. (2010). RDF Needs Annotations. In *Proceedings of the W3C Workshop on RDF Next Steps*. Palo Alto, CA, USA.

Lv, Y., Ma, Z. M. and Yan, L. (2008). Fuzzy RDF: A Data Model to Represent Fuzzy Metadata. In *FUZZ-IEEE 2008, IEEE International Conference on Fuzzy Systems, Hong Kong, China, 1-6 June, 2008, Proceedings* (pp. 1439–1445). IEEE.

Ma, Z. M. and Yan, L. (2007). Fuzzy XML data modeling with the UML and relational data models. *Data & Knowledge Engineering*, *63*(3), 972–996.

Ma, Z. M. and Yan, L. (2008). A Literature Overview of Fuzzy Database Models. *Journal of Information Science and Engineering*, *24*(1), 189–202.

Malhotra, A., Melton, J., Walsh, N. and Kay, M. (Eds.). (2010). *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2010/REC-xpath-functions-20101214/`

Manola, F. and Miller, E. (Eds.). (2004). *RDF Primer*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`

Manolescu, I., Florescu, D. and Kossmann, D. (2001). Answering XML Queries on Heterogeneous Data Sources. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao and R. T. Snodgrass (Eds.), *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy* (pp. 241–250). Morgan Kaufmann.

May, N., Helmer, S. and Moerkotte, G. (2003). Three Cases for Query Decorrelation in XQuery. In Z. Bellahsene, A. B. Chaudhri, E. Rahm, M. Rys and R. Unland (Eds.), *First International XML Database Symposium, XSym 2003* (Vol. 2824, pp. 70–84). Springer.

Mazzieri, M. (2004). A Fuzzy RDF Semantics to Represent Trust Metadata. In *1st Workshop on Semantic Web Applications and Perspectives (SWAP2004)* (pp. 83–89). Ancona, Italy.

Mazzieri, M. and Dragoni, A. F. (2005). A Fuzzy Semantics for Semantic Web Languages. In P. C. G. da Costa, K. B. Laskey, K. J. Laskey and M. Pool (Eds.), *International Semantic Web Conference, ISWC 2005, Galway, Ireland, Workshop 3: Uncertainty Reasoning for the Semantic Web, 7 November 2005* (pp. 12–22).

Mazzieri, M. and Dragoni, A. F. (2008). A Fuzzy Semantics for the Resource Description Framework. In *Uncertainty Reasoning for the Semantic Web I, ISWC International Workshops, URSW 2005-2007, Revised Selected and Invited Papers* (5327, pp. 244–261). Lecture Notes in Computer Science. Springer.

Muñoz, S., Pérez, J. and Gutiérrez, C. (2007). Minimal Deductive Systems for RDF. In E. Franconi, M. Kifer and W. May (Eds.), *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings* (Vol. 4519, pp. 53–67). Springer.

Muñoz, S., Pérez, J. and Gutiérrez, C. (2009). Simple and Efficient Minimal RDFS. *Journal of Web Semantics*, *7*(3), 220–234.

Musser, J. (2011). Open APIs & the Semantic Web: State of the Market. 2011 Semantic Technology Conference. Retrieved March 27, 2012, from `http://semtech2011.semanticweb.com/sessionPop.cfm?confid=62&proposalid=3803`

Navathe, S. B. (1992). Evolution of Data Modeling for Databases. *Communications of the ACM*, *35*(9), 112–123.

Negri, M., Pelagatti, G. and Sbattella, L. (1991). Formal Semantics of SQL Queries. *ACM Transactions on Database Systems*, *16*(3), 513–534.

Oro, E., Ruffolo, M. and Staab, S. (2010). SXPath - Extending XPath towards Spatial Querying on Web Documents. *PVLDB*, *4*(2), 129–140.

Papakonstantinou, Y., Garcia-Molina, H. and Widom, J. (1995). Object Exchange Across Heterogeneous Information Sources. In P. S. Yu and A. L. P. Chen (Eds.), *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan* (pp. 251–260). IEEE Computer Society.

Passant, A., Kopecký, J., Corlosquet, S., Berrueta, D., Palmisano, D. and Polleres, A. (2009). XSPARQL: Use cases. W3C Member Submission. Retrieved March 27, 2012, from `http://www.w3.org/Submission/2009/SUBM-xsparql-use-cases-20090120/`

Patel-Schneider, P. F. and Siméon, J. (2003). The Yin/Yang Web: A Unified Model for XML Syntax and RDF Semantics. *IEEE Transactions on Knowledge and Data Engineering*, *15*(4), 797–812.

Pérez, J., Arenas, M. and Gutiérrez, C. (2006). Semantics and Complexity of SPARQL. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, ... L. Aroyo (Eds.), *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings* (Vol. 4273, pp. 30–43). Springer.

Pérez, J., Arenas, M. and Gutiérrez, C. (2008). nSPARQL: A Navigational Language for RDF. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin and K. Thirunarayan (Eds.), *The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings* (Vol. 5318, pp. 66–81). Springer.

Pérez, J., Arenas, M. and Gutiérrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, *34*(3), 1–45.

Perry, M., Jain, P. and Sheth, A. P. (2011). SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. In N. Ashish and A. P. Sheth (Eds.), *Geospatial Semantics and the Semantic Web* (Vol. 12, pp. 61–86). Semantic Web And Beyond Computing for Human Experience. Springer.

Polleres, A. (2007). From SPARQL to Rules (and back). In *Proceedings of the 16th World Wide Web Conference (WWW2007)*. Banff, Canada.

Polleres, A., Krennwallner, T., Lopes, N., Kopecký, J. and Decker, S. (2009). XSPARQL Language Specification. W3C Member Submission. Retrieved March 27, 2012, from `http://www.w3.org/Submission/2009/SUBM-xsparql-language-specification-20090120/`

Prud'hommeaux, E. and Buil-Aranda, C. (Eds.). (2011). *SPARQL 1.1 Federated Query*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2011/WD-sparql11-federated-query-20111117/`

Prud'hommeaux, E. and Seaborne, A. (Eds.). (2008). *SPARQL Query Language for RDF*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`

Pugliese, A., Udrea, O. and Subrahmanian, V. S. (2008). Scaling RDF with time. In J. Huai, R. Chen, H.-W. Hon, Y. Liu, W.-Y. Ma, A. Tomkins and X. Zhang (Eds.), *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008* (pp. 605–614). ACM.

Rizzolo, F. and Vaisman, A. A. (2008). Temporal XML: modeling, indexing, and query processing. *VLDB Journal*, *17*(5), 1179–1212.

Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J. and Siméon, J. (Eds.). (2011). *XQuery Update Facility 1.0*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/`

Rodrigues, T., Rosa, P. and Cardoso, J. (2008). Moving from syntactic to semantic organizations using JXML2OWL. *Computers in Industry*, *59*(8), 808–819.

Schenk, S. (2008). On the Semantics of Trust and Caching in the Semantic Web. In *Proceedings of 7th International Semantic Web Conference (ISWC'2008)* (pp. 533–549).

Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I. and Busse, R. (2002). XMark: A Benchmark for XML Data Management. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China* (pp. 974–985). Morgan Kaufmann.

Farrell, J. and Lausen, H. (Eds.). (2007). *Semantic Annotations for WSDL and XML Schema*. W3C Recommendation. Retrieved March 27, 2012, from `http://www.w3.org/TR/2007/REC-sawsdl-20070828/`

Sheth, A. P. and Larson, J. A. (1990). Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, *22*(3), 183–236.

Silberschatz, A., Korth, H. F. and Sudarshan, S. (2005). *Database System Concepts, 5th Edition*. McGraw-Hill Book Company.

Silberschatz, A., Korth, H. F. and Sudarshan, S. (1996). Data Models. *ACM Computing Surveys*, *28*(1), 105–108.

Siméon, J. and Wadler, P. (2003). The Essence of XML. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 1–13).

Snodgrass, R. T. (1990). Temporal Databases - Status and Research Directions. *SIGMOD Record*, *19*(4), 83–89.

Snodgrass, R. T. (1999). *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann.

Snodgrass, R. T., Ahn, I., Ariav, G., Batory, D. S., Clifford, J., Dyreson, C. E., ... Sripada, S. M. (1994). TSQL2 Language Specification. *SIGMOD Record*, *23*(1), 65–86.

Sporny, M., Inkster, T., Story, H., Harbulot, B. and Bachmann-Gmür, R. (2011). *WebID 1.0 - Web Identification and Discovery*. W3C. Retrieved September 24, 2012, from `http://www.w3.org/2005/Incubator/webid/spec/drafts/ED-webid-20111212/`

Stephens, S. (2007). The Enterprise Semantic Web. In J. Cardoso, M. Hepp and M. D. Lytras (Eds.), *The Semantic Web: Real-World Applications from Industry* (Vol. 6, pp. 17–37). Semantic Web And Beyond Computing for Human Experience. Springer.

Stonebraker, M. and Wong, E. (1974). Access Control in a Relational Data Base Management System by Query Modification. In *Proceedings of the 1974 annual conference - Volume 1* (pp. 180–186). ACM '74. New York, NY, USA: ACM.

Straccia, U. (2009). A Minimal Deductive System for General Fuzzy RDF. In A. Polleres and T. Swift (Eds.), *Web Reasoning and Rule Systems, Third International Conference, RR 2009, Chantilly, VA, USA, October 25-26, 2009, Proceedings* (Vol. 5837, pp. 166–181). Springer.

Straccia, U., Lopes, N., Lukácsy, G. and Polleres, A. (2010). A General Framework for Representing and Reasoning with Annotated Semantic Web Data. In M. Fox and D. Poole (Eds.), *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

Suciu, D. (1998). Semistructured Data and XML. In K. Tanaka and S. Ghandeharizadeh (Eds.), *The 5th International Conference of Foundations of Data Organization (FODO'98), Kobe, Japan, November 12-13, 1998* (pp. 1–12).

Swartz, A. (2002). MusicBrainz: A Semantic Web Service. *IEEE Intelligent Systems*, *17*(1), 76–77.

Tao, J., Sirin, E., Bao, J. and McGuinness, D. L. (2010). Integrity Constraints in OWL. In M. Fox and D. Poole (Eds.), *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

Tappolet, J. and Bernstein, A. (2009). Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, ...

E. P. B. Simperl (Eds.), *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings* (Vol. 5554, pp. 308–322). Lecture Notes in Computer Science. Springer.

Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N. (Eds.). (2004). *XML Schema Part 1: Structures Second Edition*. W3C. Retrieved March 27, 2012, from `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`

Udrea, O., Recupero, D. R. and Subrahmanian, V. S. (2006). Annotated RDF. In *The Semantic Web: Research and Applications, 3rd European Semantic Web Conference, ESWC 2006* (4011, pp. 487–501). Lecture Notes in Computer Science. Springer.

Udrea, O., Recupero, D. R. and Subrahmanian, V. S. (2010). Annotated RDF. *ACM Transactions on Computational Logic*, *11*(2), 1–41.

Vrandecic, D., Dengler, F., Rudolph, S. and Erdmann, M. (2005). RDF Syntax Normalization Using XML Validation. In L. Kagal, O. Lassila and T. Finin (Eds.), *Proceedings of the SemRUs 2009: Semantics for the Rest of Us – Variants of Semantic Web Languages in the Real World* (Vol. 521). CEUR-WS.org.

Walsh, N. (2003). RDF Twig: accessing RDF graphs in XSLT. In *Proceedings of the Extreme Markup Languages, 4-8 August 2003,* Montréal, Quebec, Canada.

Widom, J. (2005). Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR* (pp. 262–276).

Wiederhold, G. (1992). Mediators in the Architecture of Future Information Systems. *IEEE Computer*, *25*(3), 38–49.

Wiederhold, G., Fries, J. F. and Weyl, S. (1975). Structured organization of clinical data bases. In *American Federation of Information Processing Societies: 1975 National Computer Conference, 19-22 May 1975, Anaheim, CA, USA* (Vol. 44, pp. 479–485). AFIPS Press.

Wielemaker, J., Huang, Z. and van der Meij, L. (2008). SWI-Prolog and the Web. *Theory and Practice of Logic Programming*, *8*(3), 363–392.

Woodruff, A. and Stonebraker, M. (1997). Supporting Fine-grained Data Lineage in a Database Visualization Environment. In W. A. Gray and P.-Å. Larson (Eds.), *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K* (pp. 91–102). IEEE Computer Society.

Yu, C. and Popa, L. (2004). Constraint-Based XML Query Rewriting For Data Integration. In G. Weikum, A. C. König and S. Deßloch (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004* (pp. 371–382). ACM.

Ziegler, P. and Dittrich, K. R. (2004). Three Decades of Data Integration - All Problems Solved? In R. Jacquart (Ed.), *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France* (pp. 3–12). Kluwer.

Zimmermann, A., Lopes, N., Polleres, A. and Straccia, U. (2012). A General Framework for Representing, Reasoning and Querying with Annotated Semantic Web Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, *11*, 72–95.

List of Figures

List of Tables

List of Data

List of Queries

List of Examples

List of Acronyms

RDF	Resource Description Framework
SQL	Structured Query Language
DTD	Document Type Definition
WWW	World Wide Web
W3C	World Wide Web Consortium
HTML	HyperText Markup Language
XML	Extensible Markup Language
JSON	JavaScript Object Notation
OEM	Object Exchange Model
LOD	Linking Open Data
URI	Uniform Resource Identifier
RDFS	RDF Schema
Infoset	XML Information Set
XSLT	XSL Transformations
XDM	XQuery 1.0 and XPath 2.0 Data Model
XSD	XML Schema
BGP	Basic Graph Pattern
BAP	Basic Annotated Pattern
XPath	XML Path Language
GRDDL	Gleaning Resource Descriptions from Dialects of Languages
SAWSDL	Semantic Annotations for Web Services Description Language
FOAF	Friend Of A Friend
OWL	Web Ontology Language
ACL	Access Control List