View-Based techniques for the efficient management of web data

Konstantinos Karanasos

UNIVERSITÉ PARIS-SUD

ÉCOLE DOCTORALE INFORMATIQUE DE PARIS-SUD (ED 427)

Laboratoire de Recherche en Informatique (LRI)

***DISCIPLINE** INFORMATIQUE*

THÈSE DE DOCTORAT

soutenue le 29/06/2012

par

Konstantinos KARANASOS

View-Based Techniques for the Efficient Management of Web Data

Directeur de thèse :	Ioana Manolescu	Inria Saclay and Université Paris Sud
Co-directeur de thèse :	François Goasdoué	Université Paris Sud and Inria Saclay

Composition du jury :

Président du jury :		
Rapporteurs :	Alin Deutsch	University of California, San Diego
	Gerhard Weikum	Max-Planck-Institüt für Informatik
Examinateurs :	Serge Abiteboul	Inria Saclay and ENS Cachan
	Christine Froidevaux	Université Paris-Sud and Inria Saclay
	Philippe Rigaux	Conservatoire National des Arts et Métiers
	Marie-Christine Rousset	Université de Grenoble

Résumé

"Techniques fondées sur des vues matérialisées pour la gestion efficace des données du web"

Konstantinos Karanasos

De nos jours, des masses de données sont publiées à grande échelle dans des formats numériques. Une part importante de ces données a une structure complexe, typiquement organisée sous la forme d'arbres (les documents du web, comme HTML et XML, étant les plus représentatifs) ou de graphes (en particulier, les bases de données du Web Sémantique structurées en graphes, et exprimées en RDF). Exploiter ces données complexes, qu'elles soient dans un format d'accès Open Data ou bien propriétaire (au sein d'une compagnie), présente un grand intérêt. Le faire de façon efficace pour de grands volumes de données reste encore un défi.

Les vues matérialisées sont utilisées depuis longtemps pour améliorer considérablement l'évaluation des requêtes. Le principe est q'une vue stocke des résultats pre-calculés qui peuvent être utilisés pour évaluer (une partie d') une requête. L'adoption des techniques de vues matérialisées dans le contexte de données du web que nous considérons est particulièrement exigeante à cause de la complexité structurelle et sémantique des données. Cette thèse aborde deux problèmes liés à la gestion des données du web basée sur des vues matérialisées.

D'abord, nous nous concentrons sur le problème de *sélection des vues pour des ensembles de requêtes RDF*. Nous présentons un algorithme original qui, basé sur un ensemble de requêtes, propose les vues les plus appropriées à matérialiser dans la base des données. Ceci dans le but de minimiser à la fois les coûts d'évaluation des requêtes, de maintenance et de stockage des vues. Bien que les requêtes RDF contiennent typiquement un grand nombre de jointures, ce qui complique le processus de sélection de vues, notre algorithme passe à l'échelle de centaines de requêtes, un nombre non atteint par les méthodes existantes. En outre, nous proposons des techniques nouvelles pour tenir compte des données implicites qui peuvent être dérivées des schémas RDF sans complexifier davantage la sélection des vues.

La deuxième contribution de notre travail concerne la *réécriture de requêtes en utilisant des vues matérialisées XML*. Nous commençons par identifier un dialecte expressif de XQuery, correspondant aux motifs d'arbres avec des jointures sur la valeur, et nous étudions des propriétés importantes de ces requêtes, y compris l'inclusion et la minimisation. En nous fondant sur ces notions, nous considérons le problème de trouver des réécritures minimales et équivalentes d'une requête exprimée dans ce dialecte, en utilisant des vues matérialisées exprimées dans le même dialecte, et nous fournissons un algorithme correct et complet à cet effet. Notre travail dépasse l'état de l'art en permettant à chaque motif d'arbre de renvoyer un ensemble d'attributs, en prenant en charge des jointures sur

la valeur entre les motifs, et en considérant des réécritures qui combinent plusieurs vues. Enfin, nous montrons comment notre méthode de réécriture peut être appliquée dans un contexte distribué, pour la dissémination efficace d'un corpus de documents XML annotés en RDF.

Mots Clefs: XML, RDF, RDFS, données du web, vues materialisées, optimisation des requêtes, réécriture de requêtes basée sur des vues, sélection des vues

Abstract

"View-Based Techniques for the Efficient Management of Web Data"

Konstantinos Karanasos

Data is being published in digital formats at very high rates nowadays. A large share of this data has complex structure, typically organized as trees (Web documents such as HTML and XML being the most representative) or graphs (in particular, graph-structured Semantic Web databases, expressed in RDF). There is great interest in exploiting such complex data, whether in an Open Data access model or within companies owning it, and efficiently doing so for large data volumes remains challenging.

Materialized views have long been used to obtain significant performance improvements when processing queries. The principle is that a view stores pre-computed results that can be used to evaluate (possibly part of) a query. Adapting materialized view techniques to the Web data setting we consider is particularly challenging due to the structural and semantic complexity of the data. This thesis tackles two problems in the broad context of materialized view-based management of Web data.

First, we focus on the problem of *view selection for RDF query workloads*. We present a novel algorithm, which, based on a query workload, proposes the most appropriate views to be materialized in the database, in order to minimize the combined cost of query evaluation, view maintenance and view storage. Although RDF query workloads typically feature many joins, hampering the view selection process, our algorithm scales to hundreds of queries, a number unattained by existing approaches. Furthermore, we propose new techniques to account for the implicit data that can be derived by the RDF Schemas and which further complicate the view selection process.

The second contribution of our work concerns *query rewriting based on materialized XML views*. We start by identifying an expressive dialect of XQuery, corresponding to tree patterns with value joins, and study some important properties for these queries, such as containment and minimization. Based on these notions, we consider the problem of finding minimal equivalent rewritings of a query expressed in this dialect, using materialized views expressed in the same dialect, and provide a sound and complete algorithm for that purpose. Our work extends the state of the art by allowing each pattern node to return a set of attributes, supporting value joins in the patterns, and considering rewritings which combine many views. Finally, we show how our view-based query rewriting algorithm can be applied in a distributed setting, in order to efficiently disseminate corpora of XML documents carrying RDF annotations.

Keywords: XML, RDF, RDFS, Web data, materialized views, query optimization, view-based query rewriting, view selection

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The idea of the World Wide Web was first conceived in 1989 [1] and the first Web page was *online* shortly after. Since then, and over the past two decades, the Web has undoubtedly shaped the digital (and not only) world as it is perceived today. In its initial version, the Web was *read-only*: very few users had the privilege to publish content, with the vast majority of users being plain consumers of this content. However, the advent of Web 2.0 would change this picture, by giving a central role to the end users, who could now both consume and produce Web content. This is the "Read-Write Web", so called by Tim Berners-Lee, the creator of the Web [2]. Various technologies were built to facilitate information sharing: blogs, Wikis, mashups, RSS feeds, video sharing websites, to name a few. More recently, the Semantic Web movement was created in an attempt to give semantics to the Web data, further promoting the interoperability and sharing of information, with a view to form "a Web of data that can be processed directly and indirectly by machines" [BLHL01]. Nowadays, the Web is used in almost all aspects of our everyday life, from creating, modifying and sharing information to building virtual societies through social networks.

1.1 Web Data

Initially, traditional relational data, as well as unstructured data (such as documents), were used on the Web. Albeit useful in various scenarios, these data models often fail to capture the need for flexibility and the irregularity of Web data. To this end, the semi-structured data model has gained a lot of popularity for Web applications. Unlike the well-structured relational data, semi-structured data do not need to abide by a strict schema, which greatly facilitates the exchange and integration of information [ABS99]. In particular, XML and RDF have been widely adopted for representing Web data, and have both been W3C standards since 1998.

XML is a flexible, generic and platform-independent data model that has become the de facto standard for exchanging data on the Web. XML data are organized in docu-

1. The original proposal is available at http://www.w3.org/History/1989/proposal.html.
2. See, for instance, http://www.mendeley.com/research/bernerslee-readwrite-web/.

ments, which can be seen as labeled, unranked trees. Moreover, XML documents are self-describing in the sense that their structure is defined by their label structure.

On the other front, RDF is the data model on which the Semantic Web is based. An RDF dataset is a graph encoded through a set of edges of the form (s, p, o), known as triples, where s, o (the $s$ubject and $o$bject of the triple, respectively) are the nodes and p ($p$roperty of the triple) the edge between them. RDF facilitates the interoperability and integration of data on the Web, by assigning to every data resource a unique identifier (URI), which can then be referenced in a triple. An important aspect of RDF is the fact that it allows to derive implicit information: RDF Schemas can be used to define hierarchies of classes and properties, and, based on these semantic relationships, data not explicitly present in the dataset can be derived through a set of reasoning rules.

1.2 Web Data Management: Main Approaches

As the popularity of the Web increased, so did the amounts of produced Web data. Indeed, XML and RDF are being used in a plethora of applications: Web sites, Web services, RSS feeds, governmental agencies, social networks, scientific data, search engines, etc. Most recently, the Linked Open Data initiative for interconnecting the publicly available RDF datasets through links further contributed to the adoption of RDF.

Efficiently managing Web data is crucial for the end users, as well as for decision support and data analytics systems of organizations. At the same time, the extreme pace at which Web data is being produced, along with its complex structure, have made the Web data management [AMR+12] a highly active area of research.

Below, we briefly outline the various techniques that have been proposed for the storage and retrieval of XML and RDF data.

XML data management Several techniques for storing XML data have focused on mapping the data to relational tables. In order to perform this mapping, knowledge based on the expected queries and structure of the data has been exploited (e.g., [DFS99, STZ+99, TVB+02]). Apart from the relational approaches, native systems dedicated to the management of XML data have also been developed [IHW01, NDM+01]. Moreover, novel join algorithms have been devised for the efficient evaluation of queries against an XML database [AKJP+02, BKS02].

RDF data management Many approaches have been proposed for storing, indexing and querying large RDF datasets, with several of them also relying on relational database back-ends. Among the most prevalent ones stand the triple table (all triples are stored in a single huge three-column table, e.g., Sesame [BKvH02]), the vertical partitioning [AMMH07] (a two-column table is created for each distinct property of the dataset) and the property tables (a table is created for each set of properties appearing together in the data, used by Jena [WSKR03]). Various indexing schemes have also been presented (e.g., Hexastore [WKB08]). Finally, RDF-3X [NW10a], a native RDF management system, also relying on the triple table, and applying aggressive indexing, specific join optimization and cardinality estimation techniques, is considered one of the most efficient RDF data management platforms.

1.3 Motivation: Web Data and Materialized Views

Despite the abundance of proposed methods for manipulating big XML and RDF datasets, it has been observed that in most cases different approaches fare well on different types of data and queries. In this context, having knowledge of the expected query workload can be of great benefit.

This is the idea behind materialized views, i.e., queries whose results are stored in the database. Materialized views can be seen as precomputed query results, which can be exploited in order to expedite query evaluation. The most important problems related to materialized views are those of view-based query rewriting, view selection and view maintenance. In the view-based query rewriting, we are given as input a query and a set of views, and attempt to evaluate the query based on the set of views. In the view selection, we are given a query workload, and we need to choose the most appropriate views to be materialized for this specific workload. These views will be subsequently used when evaluating the queries of the workload. Finally, view maintenance focuses on the problem of keeping the materialized views up-to-date as the underlying data is updated.

The aforementioned problems have been extensively studied in the relational setting [CHS02, Hal01], as well as in the XML [BOB$^+$04, CDO08, MKVZ11, PZIÖ06, TYÖ$^+$08], and, recently, in the RDF context [CL10, DCDK11, GKLM12].

View-based data management in the setting of Web (XML and RDF) data for query optimization purposes is at the core of this thesis.

1.4 Contributions (Thesis Outline)

Aiming at the efficient view-based Web data management, this thesis addresses two main problems: the view selection for RDF query workloads, and the view-based rewriting for XML queries. Below we provide an overview of how the thesis is organized, along with the main contributions of each Chapter.

Chapter 2 provides the necessary background to follow the rest of the thesis. Moreover, it discusses related works in the areas of XML, RDF and view-based data management.

Chapter 3 focuses on the problem of RDF view selection. The contributions of this Chapter can be summarized as follows:

- This is the first work to consider the problem of selecting views to be materialized, so as to enable the evaluation of the RDF queries of a given workload based exclusively on the proposed views (without accessing the initial data).
- Inspired from a relational approach, we show how our problem can be modeled as a state optimization problem, and provide search strategies and heuristics to navigate in the space of possible view configurations.
- We have devised novel ways to take into account the implicit data during the view selection process.
- We present an extensive experimental evaluation of our techniques.
- Our approach is generic and can be also applied in the relational setting for selecting materialized views.

Chapter 4 deals with the view-based query rewriting for XML queries. In particular:

- We consider a flexible query and view dialect, which is more expressive than the ones supported by the related works in the field.
- We present a rewriting algorithm which takes as input a query and a set of views and outputs a rewriting of the query based on this set of views, if one exists. Our algorithm is sound and complete, and our rewritings are expressed in a generic algebra that can be easily supported by existing execution engines.
- We discuss the properties of containment, equivalence and minimization for the expressive language we consider, and provide practical algorithms to check for these properties.
- As part of our rewriting algorithm, we provide techniques for transforming a tree pattern to another one by means of our algebra. To the best of our knowledge, ours is the first study of algebraic transformations on tree patterns.

Chapter 5 presents AnnoVIP, a distributed system we have built for the efficient management of XML documents, complimented with RDF annotations. AnnoVIP exploits materialized views in order to speedup query evaluation. To this end, the XML view-based rewriting algorithm that will be presented in Chapter 4, has been used for creating execution plans based on existing views published somewhere in the system. Such plans will then be executed in a distributed fashion over the network.

Chapter 6 provides a summary of the thesis and discusses various proposals for future work.

Chapter 2

Background and State-of-the-art

Both XML [W3C08] and RDF [www04a] were introduced as W3C recommendations in 1998, and have since become the two most prominent models for representing data on the Web. XML is considered the de facto standard for the exchange of Web data and has immediately drawn the attention of the data management community. A bit later on, the RDF data model has also gained a lot of popularity and is currently being used in a variety of applications. In this Chapter, we first discuss the XML (Section 2.1) and then the RDF data model (Section 2.2). In these Sections we also overview the main approaches that have been proposed for storing and retrieving XML and RDF data. Moreover, since the focus of this thesis is the view-based management of Web data, in Section 2.3, we discuss the main problems that are related to materialized views, along with the main results in the area.

2.1 Tree Data: XML

The Extensible Markup Language (XML) [W3C08] is a semistructured data model that has been a W3C recommendation since 1998. It is a flexible, generic and platform-independent data model and has, thus, become the standard for data exchange on the Web. Moreover, its semi-structured nature makes it suitable for integrating data that do not abide by a strict schema. In this Section, we first overview the XML data model, along with XML languages and typing (Section 2.1.1), and then discuss various approaches for storing and retrieving XML data (Section 2.1.2).

2.1.1 Data Model, Query Languages and Schemas

XML employs a tree-structured model for representing data, which we briefly present in Section 2.1.1.1. Then, in Section 2.1.1.2 we provide an overview of the most widely adopted XML query languages, whereas in Section 2.1.1.3 we discuss how an XML document can be typed.

```
<article>
  <publishInfo>
    <author>Alice</author>
    <year>2012</year>
    <country>France</country>
  </publishInfo>
  <headline>Financial Crisis</headline>
  <topic>economy</topic>
</article>
```

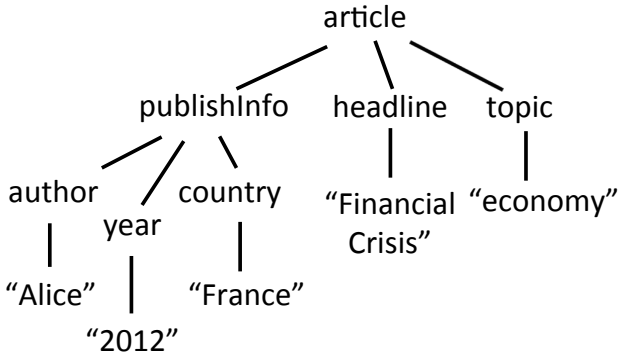


Figure 2.1: Example of an XML document.

2.1.1.1 XML Data Model

XML data is organized in documents. In particular, every XML document is a labeled (every node is assigned a label), unranked (every node can have an arbitrary number of children), ordered (there is an order between the children of each node) tree. Each node of the XML tree may be an element, a text node or an attribute. An element node may contain a list of (sub-)elements, text nodes and/or attributes as children. The text nodes contain text in Unicode, whereas the attribute nodes have a label, as well as a value. Moreover, every well-structured XML document has a single *root* element that contains all the other nodes.

A sample XML document on an article about the financial crisis is depicted in Figure 2.1. The top part of the Figure provides the serialized representation of the document, whereas the corresponding XML tree is given at the bottom part of the Figure.

2.1.1.2 XML Query Languages

Various languages have been proposed in the literature for querying XML documents. The most widely used and supported are XPath [W3C07a] and XQuery [W3C07b], which are W3C standards. The current versions of both languages rely on a common data model, namely XDM [W3C07c], and a brief description of them follows.

XPath is a language for navigating inside an XML document in order to select some of its nodes, by using path expressions. It enables the use of several axes in the path expressions, so as to facilitate the navigation in the documents, e.g., child, descen-

dant, ancestor and sibling axes. The current version of XPath is XPath 2.0 and is a syntactic subset of XQuery. XPath 2.0 extended the data model of the previous version, adding features such as intersection and complementation operators, as well as iteration capabilities. An simple example of an XPath query on the XML document of Figure 2.1 is $q = /article//country$, asking for the country that is descendant of an article in the document ($/$-edges denote parent-child relationships; $//$-edges denote ancestor-descendant ones).

XQuery is a functional language that is more powerful than XPath, and actually makes use of XPath to access specific parts of the XML documents. At the core of XQuery stand *FLOWR* expressions. The *F*or and *L*et clauses of such expressions allow to define variables that each is bound to specific nodes of the document. The *O*rder by clause is optionally used for ordering the results of the XQuery expression, where as the *W*here clause imposes various constraints on the selected by the For and Let clauses expressions. Finally, the *R*eturn clause determines the output of the expression and is capable of constructing new XML documents (and not simply return collections of values, as XPath does). Various functions (including aggregate functions), as well as user defined functions (UDFs) can be used in XQuery expressions, turning it to a Turing complete language.

Formal results on XPath and XQuery An overview of formal results on the expressiveness of several XPath 1.0 fragments, their connection to first-order logic, as well as complexity results on the evaluation of XPath are given in [BK08]. Among them, it is shown that the core navigational XPath fragment, along with the aggregate features of the language can express all first-order queries. Moreover, the combined complexity (i.e., when both data and queries are considered variable) of full XPath 1.0 evaluation is in PTIME. However, evaluating a navigational XPath 2.0 query is shown to be a PSPACE-complete problem [tCM07]. Finally, the expressiveness and complexity for various fragments of XQuery is studied in [BK09].

2.1.1.3 XML Typing

XML documents are self-describing, that is, the structure of the document is defined by its label structure. The lack of an a priori defined schema is essential for the flexibility of XML. However, albeit not compulsory, one can also specify schemas for typing XML documents, if desired.

There are two mechanisms that are widely used for XML typing: the Document Type Definition (DTD) [W3C04] and the XML Schema [XML], both of which are W3C recommendations. Among them, DTD was introduced first, and allows to determine the structure of a document (e.g., the children that each element with a specific label may have) through regular expressions. Although more complicated in terms of syntax, the XML Schema is more expressive, allowing to define constraints that were not possible through DTD (e.g., constraints on values, enumerated types). An XML document is *valid* against a schema, if it respects the constraints that are specified in the given schema. The theoretical underpinnings for XML typing are provided by tree automata [CDG+07].

2.1.2 XML Data Management

Although the in-memory processing of queries over XML documents is crucial, the increasing volumes of available XML data have led to the need of storing such data in disk. At the same time, new methods were devised for efficiently retrieving XML data stored in (disk-resident) databases. In this Section, we discuss methods for the efficient XML storage (Section 2.1.2.1) and retrieval (Section 2.1.2.1).

2.1.2.1 Storing XML Data

Many works have focused on employing relational database systems (RDBMS, in short) for the storage of XML documents. These works proposed techniques to map the XML documents to relational tables, based on some observations. Among them, in [STZ+99], knowledge from given DTDs was used to perform the mapping of XML to relations, whereas in [DFS99] the mapping was done based on knowledge for expected data and query workloads. Several other approaches also exist, e.g., [FM00, TVB+02]. In most of these systems, XML queries are translated to relational ones in order to retrieve the data.

Along with relational approaches, native XML systems have also been developed, such as the Lore [IHW01], Tukwila [MW99] and Niagara [NDM+01] systems.

Moreover, the MonetDB column-store system has been used to store and retrieve XML documents [BGvK+06].

The most important commercial RDBMS also provide support for XML, including IBM DB2 [BCH+06], Microsoft SQL Server [PCS+04] and Oracle [LM09]. Furthermore, the SQL/XML standard is an extension to the SQL language, providing the *xml* datatype, which can be used to store and retrieve XML documents.

Finally, eXist [eXi] and BaseX [Bas] are two open source XML database systems that have been widely used lately.

2.1.2.2 Retrieving XML Data

When storing XML data in a database, it is a common practice to assign a unique ID to every XML node. There have been proposed ID schemes that capture information about the position of each none in the XML document (such as the start and end point of the node, its depth in the tree, etc.), which can then be used to speed up query evaluation (e.g., efficiently determine whether a node is the parent of another). One of the most prominent ID schemes is Dewey-based IDs [TVB+02, LLCC05].

The ID schemes have been exploited in order to create novel, efficient join algorithms for the processing of XML queries in databases. More specifically, a variation of the traditional merge join algorithm, the multi-predicate merge join (MPMGJN) was first proposed in [ZND+01]. This algorithm was improved by the tree-merge and stack-tree structural join algorithms [AKJP+02], based on the idea that by sorting the join inputs (candidate ancestors and descendants), the structural join can be performed through a single pass over these inputs. The join algorithms mentioned so far need to apply a structural join for each edge (relationship between nodes) of the XML query. To circumvent this problem

and further expedite query evaluation, the holistic twig join algorithm [BKS02] builds the result of the query in a single pass over all the input relationships in parallel, eliminating the need for storing and sorting intermediate results.

Distributed XML data management A collection of works has focused on the problem of answering queries over a global XML database published over a peer-to-peer overlay network, e.g., [BC06, AMP+08, LP08, MKK08, KKMZ12, KKMZ11]. Recently, the data management of XML documents on the Cloud has also been addressed [KCS11, CRCM12].

2.2 Graph Data: RDF

The Resource Description Framework (RDF) [www04a] is a graph data model, which has been recommended by W3C since 1998. Originally designed as a data model for metadata, RDF has become the cornerstone of the Semantic Web [BLHL01] for describing information about (Web) resources, in a machine-exploitable way. Its graph-structured, schema-less nature, along with its capability of expressing implicit information, make it suitable for expressing heterogeneous, irregular data, and has thus been used in a variety of applications. In this Section, we first present the RDF graphs and queries (Section 2.2.1), then we describe how implicit information is modeled in RDF (Section 2.2.2), and finally we discuss the proposed approaches for managing RDF data (Section 2.2.3).

2.2.1 RDF Graphs and Queries

RDF graphs An RDF statement about a resource is expressed as a *triple* of the form (s, p, o), stating that a *subject* s has the corresponding *property* p, and the value of that property is the *object* o. A set of RDF triples forms an RDF *graph*.

Given a set U of *Uniform Resource Identifiers* [1] (URIs), a set L of literals (constants), and a set B of blank nodes (*unknown* URIs or literals), such that U, B and L are pairwise disjoint, a triple is well-formed whenever its subject belongs to $U \cup B$, its property belongs to U, and its object belongs to $U \cup B \cup L$. In the following, we only consider well-formed triples.

Blank nodes are essential features of RDF allowing to support incomplete information. For instance, one can use a blank node _:b_1 to state that the country of _:b_1 is *France* while the city of the same _:b_1 is *Brest*.

Figure 2.2 shows how to use triples to describe resources. Observe that the name rdf represents the RDF namespace [2], which is used when writing the URIs of classes and

1. Uniform Resource Identifiers provide naming schemes for referring to resources using keys, in the usual database sense.

2. A namespace is a URI used to group resources. When a namespace is given a `name`, a `resource` within that namespace can be simply written `name:resource`. For instance, `inria:oak` refers to resource `oak` within the namespace `inria`; it corresponds to the URI `http://team.inria.fr/oak` whenever the namespace `http://team.inria.fr/` is named `inria`.

Constructor	Triple
Class assertion	$(s, \text{rdf:type}, o)$
Property assertion	(s, p, o)

Figure 2.2: RDF statements.

$\{(mus{:}vangogh, mus{:}name, \text{“}VanGogh\text{”}),$
$(mus{:}vangogh, rdf{:}type, mus{:}painter),$
$(mus{:}vangogh, mus{:}hasPainted, mus{:}painting1),$
$(mus{:}painter, rdfs{:}subClassOf, mus{:}artist)\}$

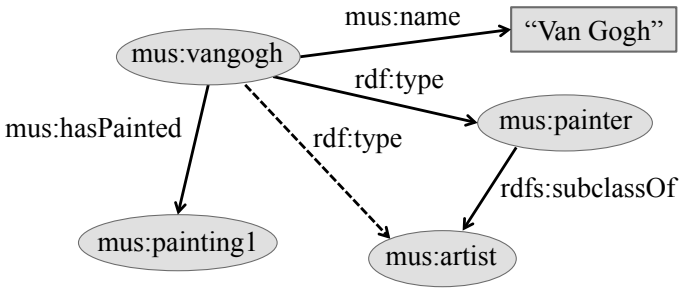


Figure 2.3: Example of an RDF graph.

properties comprised in the RDF standard.

A more intuitive representation of an RDF graph can be drawn from its triples. Every (distinct) subject or object value appearing in some triple is represented by a node labeled with this value. For each triple, there is a directed edge from the subject node to the object node, labeled with the property value. An example of an RDF graph is given in Figure 2.3. The top part of the Figure provides the triple representation, whereas the bottom part the sagittal representation, of the RDF graph. The rectangles represent URIs and the ovals represent literals. Moreover, the name mus denotes a namespace that is used for our example.

RDF queries The official language for expressing RDF queries is SPARQL [SPA], and has been recommended by W3C since 2008. In this thesis, we consider the well-known subset of SPARQL, consisting of *basic graph pattern* (BGP) queries. A BGP is a *set of triple patterns*, or triples in short. Each triple has a subject, property and object. Subjects and properties can be URIs, blank nodes or variables; objects can also be literals.

BGP queries can be expressed as conjunctive queries over a table $t(s, p, o)$ that contains all triples of the RDF graph. As an example, consider the following query, which asks for the name of someone who is a painter, as well as his paintings:

$$q(Y, Z){:-}t(X, mus{:}name, Y), t(X, rdf{:}type, mus{:}painter), t(X, mus{:}hasPainted, Z)$$

Constructor	Triple
Subclass constraint	$(s, \text{rdfs:subClassOf}, o)$
Subproperty constraint	$(s, \text{rdfs:subPropertyOf}, o)$
Domain typing constraint	$(s, \text{rdfs:domain}, o)$
Range typing constraint	$(s, \text{rdfs:range}, o)$

Figure 2.4: RDFS statements expressing semantic constraints between classes and properties.

2.2.2 Entailment of Implicit Information

The W3C names *RDF entailment* the mechanism through which, based on the set of explicit triples and some *entailment rules* (to be described shortly), *implicit triples* are derived. A triple (s, p, o) is entailed by an RDF graph, if and only if there is a sequence of applications of entailment rules that leads from the graph to (s, p, o), where at each step of the entailment sequence, the triples previously entailed are also taken into account.

Some implicit triples are obtained by generalizing existing triples using blank nodes. For instance, a triple (s, p, o) entails the triple $(\_{:}b, p, o)$, where s is a URI and $\_{:}b$ denotes a blank node. Thus, from the triple $(mus{:}vangogh, rdf{:}type, mus{:}painter)$, we can derive the triple $(\_{:}b_1, rdf{:}type, mus{:}painter)$, where $\_{:}b_1$ is a blank node.

Various other entailment rules exist, such as the ones that use the rdfs:Class URI to denote the class to which a resource may belong. Among the existing rules, of particular interest are the ones derived from an *RDF Schema* [www04d] (RDFS, in short). RDFS is a valuable feature of RDF that allows enhancing the descriptions in RDF graphs. It declares *semantic constraints* between the classes and the properties used in graphs. Figure 2.4 shows the allowed constraints and how to express them. The name rdfs denotes the RDFS namespace that is used when writing the URIs of classes and properties comprised in the RDFS standard. For instance, considering the example of Figure 2.3, we know that $mus{:}vangogh$ is a painter ($mus{:}painter$), and moreover, the painter is a subclass of the artist ($mus{:}artist$). To this end, we can derive the implicit triple $(mus{:}vagogh, rdf{:}type, mus{:}artist)$, which is depicted with a dashed line in the Figure.

Taking into account implicit triples in query answering is necessary for guaranteeing the completeness of query results. To this respect, two are the main techniques that have been proposed: database saturation and query reformulation. The former approach adds in the database all implicit triples specified in the RDF recommendation [www04a], whereas the latter reformulates a query into a union of queries in order to account for the implicit triples, while leaving the database intact [AGR07, CGL+07].

More details about the RDF query answering based both on explicit and implicit triples are given in Chapter 3 of this thesis.

2.2.3 RDF Data Management

Along with the abundance of available RDF datasets came the interest of the data management community that has so far proposed a wide variety of approaches for efficiently storing and querying large RDF graphs, the most prominent of which are described below.

Some of the earliest works in the field are Jena [WSKR03] and Sesame [BKvH02]. Jena stores the RDF data in *property tables*, that is, properties that often appear together in the dataset are clustered and stored in the same relational table. On the other hand, Sesame uses the *triple table*, storing all RDF triples in a single, huge, 3-attribute relational table. However, it has been experimentally shown (e.g., [AMMH07]) that both these systems suffer from scalability issues when faced with large RDF datasets.

In vertical partitioning [AMMH07] one 2-attribute relation (storing the subject and the object of the triples) is created for each property value. One of the shortcomings of this approach is that all tables need to be accessed for queries with variables in the property position. Vertical partitioning has been tested using both row- and column-store relational systems [SGK+08].

Hexastore [WKB08] follows the triple table approach, but uses aggressive 6-way indexing (indexes are built for each permutation of the attributes of the triple table), significantly improving query execution times.

The RDF-3X system [NW10a] is considered one of the most efficient ones for manipulating RDF datasets. It also relies on the triple table, and employs the aggressive indexing of Hexastore, specific join optimization techniques (such as Sideways Information Passing), as well as join ordering based on more accurate selectivity estimations [NW08, NW09]. An extension of RDF-3X, called x-RDF-3X, added to the system support for online updates, versioning and transactions [NW10b].

The gStore system [ZMC+11] is one of the few systems (along with, e.g., Grin [UPS07]) that avoids using a relational approach for storing RDF data. Instead, it follows a graph-based and employs a novel index, with a view to support frequent updates in the data, as well as queries containing regular expressions with wildcards.

Note that most of the existing systems use *dictionary encoding*, substituting the string values of RDF triples with fixed-width identifiers.

RDF support has been added to commercial systems as well, including Oracle [CDES05], IBM DB2 and Virtuoso [Sof].

Distributed RDF query processing An early work on processing conjunctive RDF queries over structured peer-to-peer networks based on a distributed hash table (DHT, in short) was proposed in [LIK06]. Most recently, a scale-out architecture for scalable RDF data management, exploiting state-of-the-art RDF stores and the MapReduce framework [DG04] was introduced in [HAR11]. Distributed cloud-based RDF data management using the Amazon Web Services was presented in [BGKM12].

Limitations of RDF data management systems Despite the significant number of existing works for managing RDF data, almost none of them actively takes into consideration the implicit triples brought by the RDF Schemas. In contrast, it is assumed that implicit information is already available through database saturation. Moreover, there is a limited support for the full SPARQL language. For example, queries seeking for triples that are connected via a path of arbitrary length are not supported, although they may be of practical interest for many applications (e.g., social network graphs). This feature, called property paths, was introduced in the specification of SPARQL 1.1 (see [ACP12] for more details on property paths).

In a recent study on the characteristics of various RDF datasets [DKSU11], the metric of *structuredness* was introduced on an attempt to quantify how structured a dataset is (i.e., how close to structured relational data it is). Such metrics could be used for better tuning the storage, indexing and querying techniques for RDF datasets.

2.3 Materialized Views

A materialized view is a query whose results (called *view extent*) are stored in the database. Views are defined over some base relations (or documents in a document-oriented database, such as XML databases), which constitute the *base* data. Due to their multiple benefits, materialized views have been used in a variety of applications and have, thus, been extensively studied in the database literature. Some of the most prevalent such applications are the following:

Query optimization Materialized views can be seen as precomputed query results. To this end, they can considerably expedite query evaluation, since part of the computation for evaluating the query was performed during the materialization of the view, and can thus be avoided at execution time. Employing materialized view at the context of query optimization is at the core of the present thesis.

Data warehouse design A data warehouse is a repository of integrated data, coming from multiple, possibly distributed and heterogeneous data sources. The information stored in a data warehouse can be used for decision support. In this context, materialized views have been used to optimize the evaluation of such decision support queries.

Data placement over a network When users are distributed over a network, query response times may be high due to the need for data transfer from the data holders to the end users. In this case, materialized views can also be beneficial, by placing the data needed by its user in a way that reduces query response time.

In what follows, we present the three basic problems pertinent to materialized views: answering queries using views (Section 2.3.1), view selection (Section 2.3.2) and view maintenance (Section 2.3.3). In Section 2.3.4, we briefly discuss some additional problems related to materialized views.

2.3.1 Answering Queries Using Views

Given a query q and a set of views $\mathcal{V}$, *answering queries using views*, also known as *view-based query rewriting*, deals with the problem of how can q be answered using the views in $\mathcal{V}$.

Query rewriting in relational databases A comprehensive survey on answering queries using views in relational databases is given in [Hal01]. Rewriting algorithms in the context of query optimization have been proposed both for System-R style optimizations (e.g., [CKPS95]) and for transformational optimizers (e.g., [DPT99, GL01]). For data

integration scenarios, the most important works include the bucket algorithm [LRO96], the inverse-rules algorithm [DGL00] and the MiniCon algorithm [PH01].

Results on the complexity of the problem of answering queries using materialized views are reported in [AD98].

Query rewriting in XML As this is the topic of Chapter 4, a detailed discussion on the XML view-based query rewriting methods is given in Section 4.9.

Query rewriting in RDF Since the conjunctive subset of SPARQL (the BGP queries) can be mapped to relational conjunctive queries, well-known relational algorithms can be used to rewrite such queries. Recently, a native query rewriting algorithm under set semantics for a fragment of SPARQL was proposed in [LDK+11]. Since the output of the rewriting is a union of (conjunctive) queries of potentially exponential size, optimizations were discussed to minimize each query of the union, eliminate queries with empty results and prune the search space of empty rewritings.

2.3.2 View Selection Problem

Given a query workload $\mathcal{Q}$, *view selection* deals with the problem of choosing the most suitable view set $\mathcal{V}$ to be materialized in order to answer the queries in $\mathcal{Q}$. The properties that $\mathcal{V}$ should exhibit (and possible requirements it should meet) depend on the specific application. Such properties are commonly associated with the query evaluation time (using the proposed views), the view storage space and the view maintenance cost. In some cases the view storage space is given as a bound to the problem.

View selection in relational databases View selection has been extensively studied, especially in the context of data warehouses for SPJ queries [TS97] and OLAP queries [Gup97, HRU96, GM05]. Several formal results concerning the view selection problems are provided in [CHS02].

View selection in XML View selection has been addressed for XPath [MS05, TYT+09], as well as for XQuery dialects [KMV12].

View selection in RDF Our work [GKLM12], presented in Chapter 3 of this thesis, is among the first to explore the problem of view selection in RDF databases. Closely related works are [GKLM12] are [CL10] and [DCDK11]. RDFMatView [CL10] recommends RDF indices to materialize for a given workload, while in [DCDK11] a set of path expressions appearing in the given workload is selected to be materialized, both aiming at improving the performance of query evaluation.

2.3.3 View Maintenance

Since the content of the materialized views relies on some base data, *view maintenance* deals with the problem of keeping the view data up-to-date while the corresponding base data is updated.

Views can be updated by recomputing the view upon each update. As this technique is in most cases highly inefficient, *incremental* view maintenance is widely adopted, that

is, only the needed changes are applied to the view, based on the update of the base data.

The problem of incremental view maintenance has been studied for relational databases (e.g., [GMS93, CGL+96, SBCL00]), VLSD (Very Large Scale Distributed) shared-nothing databases (e.g., [ASC+09]), as well as XML databases (e.g., [BGMS11, STP+05]). The approaches that have been proposed for view maintenance can be categorized as follows:

Immediate (eager) vs. deferred (lazy) In the immediate maintenance, views are updated as part of the transaction that caused the update of the base tables. Clearly, this imposes a significant overhead on update transactions, especially in the existence of a large number of views. Contrariwise, in the deferred maintenance, views are not updated as part of the update transaction; the update of the views can occur later on.

Pre-update state vs. post-update state Incremental view maintenance relies on a set of *incremental* queries, which are issued on the base tables to compute the changes that need to be applied to the views. These queries can be evaluated either on the pre-update or the post-update state of the base tables. In the immediate maintenance we have still access to the pre-update state, which makes the maintenance algorithms simpler. However, this is not the case for the deferred maintenance, where more sophisticated algorithms are needed.

Push vs. pull propagation of updates This categorization is related to the way updates are propagated to the views. In the push mode, eager maintenance is required, whereas the pull mode can be coupled either with eager or deferred propagation of the updates.

2.3.4 Problems Related to Materialized Views

Query containment and equivalence The notions of query containment and equivalence enable the comparison between different reformulations of queries, and are crucial for the view-related problems discussed in the previous sections.

Given a query q and a database D, by $q(D)$ we denote the result of evaluating q over D. Let q_1, q_2 be two queries. Query q_1 is contained in q_2, denoted $q_1 \sqsubseteq q_2$, if for any database D, we have $q_1(D) \subseteq q_2(D)$. The two queries are equivalent, denoted $q_1 \equiv q_2$, if for any database D, we have $q_1(D) = q_2(D)$.

Containment and equivalence have been studied for conjunctive queries under set (e.g., [CM77, KV00]) and bag semantics [CV93, JKV06], as well as for XPath [MS04] and SPARQL [LPPS12] queries. New results on containment and equivalence for the XQuery fragment we consider, are given in Chapter 4.

Multi-query optimization Given a query workload, multi-query optimization (MQO, in short) attempts to find common sub-expressions between the queries, which can be evaluated only once and then be re-used by the various queries in order to speed up query evaluation. The process of finding common subexpressions makes the problem closely related to the view selection (although the goal of MQO is not to materialize the common subexpressions at the end). MQO is a crucial problem for DBMS and has been the focus

of research for many years, with [Sel88] being one of the earliest and [ZLFL07] one of the most recent works in the field.

The MQO problem for SPARQL query workloads was addressed in [LKDL12]. In this work, the workload is partitioned in groups and techniques for finding common sub-expressions among each group are considered.

2.4 Summary

In this Chapter we presented two of the most popular data models for representing and sharing data on the Web, namely XML and RDF, along with the main approaches that have been proposed by the data management community for the efficient storage and retrieval of such data. The large amounts of available XML and RDF data nowadays, along with its complex (tree and graph) structure still raise significant challenges for their efficient manipulation. To this end, we have discussed the management of data based on materialized views, which has been widely used for query optimization in many contexts, and which can thus be greatly beneficial also when employed in the context of the complex Web data we consider.

Chapter 3

View Selection in Semantic Web Databases

In this Chapter, we consider the setting of a Semantic Web database, containing both explicit data encoded in RDF triples, and implicit data, implied by the RDF semantics. Based on a query workload, we address the problem of selecting a set of views to be materialized in the database, minimizing a combination of query processing, view storage, and view maintenance costs. Starting from an existing relational view selection method, we devise new algorithms for recommending view sets, and show that they scale significantly beyond the existing relational ones, when adapted to the RDF context. To account for implicit triples in query answers, we propose a novel RDF query reformulation algorithm and an innovative way of incorporating it into view selection in order to avoid a combinatorial explosion in the complexity of the selection process. The interest of our techniques is demonstrated through a set of experiments.

The work described in this Chapter has led to an early publication in the French national database conference [GKLM10b] (without formal proceedings). The prototype we built was demonstrated in [GKLM10a, GKLM11]. The final version of this work, which the present Chapter closely follows, appeared in [GKLM12].

3.1 Motivation and outline

A key ingredient for the Semantic Web vision [BLHL01] is a data format for describing items from the real and digital world in a machine-exploitable way. The W3C's Resource Description Framework [www04a] (RDF, in short) is a leading candidate for this role.

At a first look, querying RDF resembles querying relational data. Indeed, at the core of the W3C's SPARQL query language for RDF [www08] lies conjunctive relational-style querying. There are, however, several important differences in the RDF data model. First, an RDF dataset is a single large set of triples, in contrast with the typical relational database featuring many relations with varying numbers of attributes. Second, RDF triples may feature *blank nodes*, standing for unknown constants or URIs; an RDF

database may, for instance, state that the *author* of X is *Jane* while the *date* of X is *4/1/2011*, for a given, unknown resource X. This contrasts with standard relational databases, where all attribute values are either constants or $null$. Finally, in typical relational databases, all data is *explicit*, whereas the semantics of RDF entails a set of *implicit* triples which must be reflected in query answers. One important source of implicit triples follows from the use of an (optional) RDF Schema [www04a] (or RDFS, in short), to enhance the descriptive power of an RDF dataset. For instance, assume the RDF database contains the fact that the *driverLicenseNo* of *John* is *12345*, whereas an RDF Schema states that only a *person* can have a *driverLicenseNo*. Then, the fact that *John* is a *person* is implicitly present in the database, and a query asking for all *person* instances in the database must return *John*.

The complex, graph-structured RDF model is suitable for describing heterogeneous, irregular data. However, it is clearly not a good model for storing the data. Existing RDF platforms, therefore, assume a simple (application-independent) storage model, based on relational approaches in most cases, complemented by indexes and efficient query evaluation techniques [AMMH07, NW08, NW09, NW10b, SGK$^+$08, WKB08], or by RDF materialized views [CL10, DCDK11]. While indexes or views speed up the evaluation of the fragments of queries matching them, the query processor may still need to access the main RDF database to evaluate the remaining fragments of the queries.

We consider the problem of *choosing a (relational) storage model for an RDF application*. Based on the application workload, we seek a set of views to materialize over the RDF database, such that *all workload queries can be answered based solely on the recommended views, with no need to access the database*. Our goal is to enable three-tier deployment of RDF applications, where clients do not connect directly to the database, but to an application server, which could store only the relevant views. Alternatively, if the views are stored at the client, no connection is needed and the application can run off-line, independently from the database server.

RDF datasets can be very different: data may be more or less structured [DKSU11], schemas may be complex, simple, or absent, updates may be rare or frequent. Moreover, RDF applications may differ in the shape, size and similarity of queries, costs of propagating updates to the views, etc. To capture this variety, we characterize candidate view sets by a cost function, which combines (i) query evaluation costs, (ii) view maintenance costs and (iii) view storage space. Our contributions are the following:

1. This is the first study of RDF materialized view selection supporting the rewriting of all workload queries. We show how to model this as a search problem in a space of states, inspired from a previous work in relational data warehousing [TLS01].
2. Implicit triples entailed by the RDF semantics [www04a] must be reflected in the recommended materialized views, since they may participate to query results. Two methods are currently used to include implicit tuples in query results. *Database saturation* adds them to the database, while *query reformulation* leaves the database intact and modifies queries in order to also capture implicit triples. Our approach requires no special adaptation if applied on a saturated database. For the reformulation scenario, we propose a novel RDF query reformulation algorithm. This algorithm extends the state of the art in query processing in the presence of RDF

Schemas [AGR07, CGL+07], and is a contribution that can be applied beyond the context of this work. Moreover, we propose an innovative method of using reformulation (called *post-reformulation*), which enables us to efficiently take into account implicit triples in our view selection approach.

3. We consider efficient search strategies, given that the complexity of complete search is extremely high. Existing strategies for relational view selection [TLS01] grow out of memory and fail to produce a solution when the number of atoms in the query workload grows. Since RDF atoms are short (just three attributes), RDF queries are syntactically more complex (they have more atoms) than relational queries retrieving the same information, making this scale problem particularly acute for RDF. We propose a set of new strategies and heuristics which greatly improve the scalability of the search. Our strategies can be used in the relational setting as well.
4. We study the efficiency and effectiveness of the above algorithms, and their improvement over existing similar approaches, through a set of experiments.

This Chapter is organized as follows. Section 3.2 formalizes the problem we consider. Section 3.3 presents the view selection problem as a search problem in a space of candidate states, whereas Section 3.4 discusses the inclusion of implicit RDF triples in our approach. Section 3.5 describes the search strategies and heuristics used to navigate in the search space. Then, Section 3.6 outlines the we built to implement our view selection approach, and Section 3.7 presents our experimental evaluation. Section 3.8 discusses related works, then we conclude.

3.2 Problem Statement

In accordance with the RDF specification [www04a], we view an RDF database as a set of (s, p, o) triples, where s is the *subject*, p the *property*, and o the *object* of the triple. RDF triples are *well-formed*, that is: subjects can be URIs or *blank nodes*, properties are URIs, while objects can be URIs, blank nodes, or literals (i.e., values). Blank nodes are placeholders for unknown *constants* (URIs or literals); from a database perspective, they can be seen as existential variables in the data. While relational tuples including a $null$ value, commonly used to represent missing information, cannot be joined ($null$ does not satisfy any predicate), RDF triples referring to *the same* blank node may be joined to construct complex results, as exemplified in Section 3.1. Due to blank nodes, an RDF database can be seen as an incomplete relational database consisting of a single *triple table* $t(s, p, o)$, under the open-world assumption [AHV95]. More details about the RDF data model are given in Chapter 2 of this thesis.

To express RDF queries (and views), we consider the basic graph pattern (BGP, in short) queries of SPARQL [www08], represented *wlog* as a special case of conjunctive queries: conjunctions of atoms, the terms of which are either free variables (also known as head variables), existential variables, or constants. We assume set semantics both for the triple table and for the results of our views and queries. Moreover, we do not use a specific representation for blank nodes in queries, although SPARQL does, because they behave exactly like existential variables.

Definition 3.2.1 (RDF queries and views)**.** *An RDF query (or view) is a conjunctive query over the triple table* $t(s, p, o)$.

We consider *wlog* queries *without Cartesian products*, i.e., each triple shares at least one variable (joins at least) with another triple. We represent a query with a Cartesian product by the set of its independent sub-queries. Finally, we assume queries and views are *minimal*, i.e., the only containment mapping from a query (or view) to itself is the identity [CM77].

As a running example, we use the following query q_1, which asks for painters that have painted "Starry Night" and having a child that is also a painter, as well as the paintings of their children:

$$q_1(X, Z) :- t(X, hasPainted, starryNight), t(X, isParentOf, Y), \\ t(Y, hasPainted, Z)$$

Based on views, one can rewrite the workload queries:

Definition 3.2.2 (Rewriting)**.** *Let* q *be an RDF query and* $V = \{v_1, v_2, \ldots, v_k\}$ *be a set of RDF views. A rewriting of* q *based on* V *is a conjunctive query* (i) *equivalent to* q *(i.e., on any dataset, it yields the same answers as* q*),* (ii) *involving only relations from* V*, and* (iii) *minimal, in the sense mentioned above.*

We are now ready to define our view selection problem, which relies on candidate view sets:

Definition 3.2.3 (Candidate view set)**.** *Let* Q *be a set of RDF queries. A candidate view set for* Q *is a pair* $\langle V, R\rangle$ *such that:*

- V *is a set of RDF views,*
- R *is a set of rewritings such that:* (i) *for every query* $q \in Q$*, there exists exactly one rewriting* $r \in R$ *of* q *using the views in* V*;* (ii) *all* V *views are useful, i.e., every view* $v \in V$ *participates in at least one rewriting* $r \in R$*.*

We consider a *cost estimation function* c^ϵ, which returns a quantitative measure of the costs associated to a view set. The lower the cost, the better the candidate view set is. Our cost components account for the effort to evaluate the view-based query rewritings, the total space occupancy of the views and the view maintenance costs as data changes. More details about c^ϵ are provided in Section 3.3.4.

Definition 3.2.4 (View selection problem)**.** *Let* $Q = \{q_1, q_2, \ldots, q_n\}$ *be a set of RDF queries and* c^ϵ *be a cost estimation function. The view selection problem consists in finding a candidate view set* $\langle V, R\rangle$ *for* Q *such that, for any other candidate view set* $\langle V', R'\rangle$ *for* Q*:* $c^\epsilon(\langle V, R\rangle) \leq c^\epsilon(\langle V', R'\rangle)$.

3.3 The Space of Candidate View Sets

This Section describes our approach for modeling the space of possible candidate view sets. Section 3.3.1 introduces the notion of a state to model one such set, while Section 3.3.2 presents a set of transitions that can be used to transform one state to another. Then, Section 3.3.3 discusses the details of detecting view and state equivalence and, finally, Section 3.3.4 shows how to assign a cost estimation to each state.

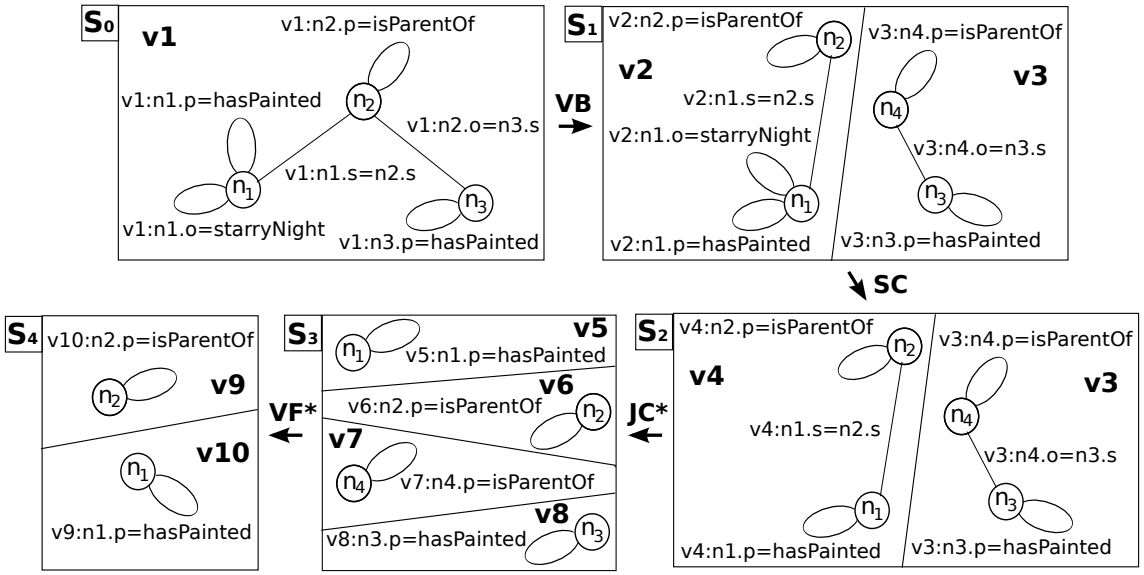


Figure 3.1: Sample initial state graph S_0, and states attained through successive transitions.

3.3.1 States

We use the notion of *state* to model a candidate view set together with the rewritings of the workload queries based on these views. The set of all possible candidate view sets is then modeled as a set of states, which we adapt from a previous work on materialized view selection in a relational data warehouse [TLS01]. From here forward, given a workload Q, we may use $S(Q)$ (possibly with subscripts or superscripts) to denote a candidate view set for Q. To ease the exposition, we also employ from [TLS01] a visual representation of each state by means of a *state graph*.

Definition 3.3.1 (State graph)**.** *Given a query set Q and a state $S_i(Q) = \langle V_i, R_i \rangle$, the state graph $G(S_i) = (N_i, E_i)$ is a directed multigraph such that:*

- *each triple t_i appearing in a view $v \in V_i$ is represented by a node $n_i \in N_i$;*
- *let t_i and t_j be two triples in a view $v \in V_i$, and a join between their attributes $t_i.a_i$ and $t_j.a_j$ (where $a_i, a_j \in \{s, p, o\}$). For each such join, there is an edge $e_i \in E_i$ connecting the respective nodes $n_i, n_j \in N_i$ and labeled $v{:}n_i.a_i = n_j.a_j$. We call e_j a* join edge*;*
- *let t_i be a triple in a view $v \in V_i$ and $n_i \in N_i$ be its corresponding node. For every constant c_i, that appears in the attribute $a_i \in \{s, p, o\}$ of t_i, an edge labeled $v{:}n_i.a_i = c_i$ connects n_i to itself. Such an edge is called* selection edge*.*

The *graph of* v is defined as the subgraph of $G(S_i)$ corresponding to v. Observe that in a view, two nodes may be connected by several join edges if their corresponding atoms are connected by more than one join predicates.

We define two states to be *equivalent* if they have the same view sets. Furthermore,

to avoid a blow-up in the storage space required by the views, *we do not consider views including Cartesian products*. In a relational setting, some Cartesian products, e.g., between small dimension tables in an OLAP context, may not raise performance issues. In contrast, in the RDF context where all data lies in a single large triple table, Cartesian products are likely not interesting queries, and their storage overhead is prohibitive. The absence of Cartesian products from our views entails that *the graph of every view is a connected component of the state graph*.

As a (simple) example, consider the state $S_0(Q) = \langle\{v_1\}, R_0\rangle$, where $Q = \{q_1\}$ is a workload containing only the previously introduced query q_1, and $v_1 = q_1$. The rewriting set R_0 consists of the trivial rewriting $\{q_1 = v_1\}$. The graph $G(S_0)$ is depicted at the upper left part of Figure 3.1, and since it corresponds to a single view, it comprises only one connected component.

3.3.2 State Transitions

To enumerate candidate view sets (or, equivalently, states), we use four transitions, inspired from [TLS01] (the differences between our transition set and the one of [TLS01] are outlined in Section 3.8). As we show in Section 3.5.1, our transition set is complete, i.e., all possible states for a given workload can be reached through our four transitions. The first three transitions remove predicates from views, thus can be seen as "relaxing", and may split a view in two, increasing the number of views. The last one factorizes two views into one, thus reducing the number of workload views. The graphs corresponding to the states before and after each transition are illustrated in Figure 3.1.

We use $v{:}e$ to denote an edge e belonging to the view v in a state graph. While we define rewritings as conjunctive queries, for ease of explanation, we will denote rewritings by (equivalent) relational algebra expressions. We use σ_e to denote a selection on the condition attached to the edge e in a view set graph. Since the query set Q is unchanged across all transitions, we omit it for readability.

Definition 3.3.2 (View Break (VB)). *Let $S = \langle V, R\rangle$ be a state, v a view in V and N_v the set of nodes of the graph of v with $|N_v| > 2$. Let N_{v_1}, N_{v_2} be two subsets of N_v such that:*

- *$N_{v_1} \not\subseteq N_{v_2}$ and $N_{v_2} \not\subseteq N_{v_1}$;*
- *$N_{v_1} \cup N_{v_2} = N_v$;*
- *the subgraph of the graph of v defined by N_{v_1} (respectively, by N_{v_2}) and the edges between these nodes is connected.*

We create two new views, v_1 and v_2. View v_1 (respectively, v_2) derives from the graph of v by copying the nodes corresponding to N_{v_1} (N_{v_2}) and the edges between them. The head variables of v_1 (v_2) are those of v appearing also in the body of v_1 (v_2), together with all additional variables appearing in the nodes $N_{v_1} \cap N_{v_2}$.

The new state $S' = \langle V', R'\rangle$ consists of:

- *$V' = (V \setminus \{v\}) \cup \{v_1, v_2\}$,*
- *$G(S')$ is obtained from $G(S)$ by removing the graph of v and adding those of v_1 and v_2, and*
- *R' is obtained from R by replacing all the occurrences of v, with $\pi_{head(v)}(v_1 \bowtie v_2)$, where $\bowtie$ is the natural join.*

For example, we apply a view break on the view v_1 of state S_0 introduced in the previous Section, and obtain the new state S_1:

$$S_1 = \langle \{v_2, v_3\}, \{q_1 = \pi_{head(v_1)}(v_2 \bowtie v_3)\} \rangle$$

The two newly introduced views v_2 and v_3 are the following:

$v_2(X, Y)$:$-t(X, hasPainted, starryNight), t(X, isParentOf, Y)$

$v_3(X, Y, Z)$:$-t(X, isParentOf, Y), t(Y, hasPainted, Z)$

Definition 3.3.3 (Selection Cut (SC)). *Let $S = \langle V, R \rangle$ be a state and v:e be a selection edge in $G(S)$. A* selection cut *on e yields a state $S' = \langle V', R' \rangle$ such that:*

- *V' is obtained from V by replacing v with a new view v', in which the constant in the selection edge e has been replaced with a fresh head variable (i.e., the fresh variable is returned by v', along with the variables returned by v),*
- *$G(S')$ is obtained from $G(S)$ by removing the graph of v and adding the one of v', and*
- *R' is obtained from R by replacing all occurrences of v with the expression $\pi_{head(v)}(\sigma_e(v'))$.*

For instance, we apply a selection cut on the edge labeled v_2:$n_1.o = starryNight$ of $G(S_1)$ and obtain the state S_2, in which v_2 is replaced by a new view v_4:

$$S_2 = \langle \{v_3, v_4\}, \{q_1 = \pi_{head(v_1)}(\pi_{head(v_2)}(\sigma_{n_1.o=starryNight}(v_4)) \bowtie v_3)\} \rangle$$

View v_4 is the following:

$v_4(X, Y, W)$:$-t(X, hasPainted, W), t(X, isParentOf, Y)$

Definition 3.3.4 (Join Cut (JC)). *Let $S = \langle V, R \rangle$ be a state and v:e be a join edge in $G(S)$ of the form $n_i.a_i = n_j.a_j$, such that $a_i, a_j \in \{s, p, o\}$. A* join cut *on e yields a state $S' = \langle V', R' \rangle$, obtained as follows:*

1. *If the graph of v is still connected after the cut, V' is obtained from V by replacing v with a new view v' in which the variable corresponding to the join edge e becomes a head variable, and the occurrence of that variable corresponding to $n_i.a_i$ is replaced by a new fresh head variable. The new rewriting set R' is obtained from R by replacing v by $\pi_{head(v)}(\sigma_e(v'))$. The new graph $G(S')$ is obtained from $G(S)$ by removing the graph of v and adding the one of v'.*
2. *If the graph of v is split in two components, V' is obtained from V by replacing v with two new symbols v'_1 and v'_2, each corresponding to one component. In each of v'_1 and v'_2, the join variable of e becomes a head variable. The new rewriting set R' is obtained from R by replacing v by $\pi_{head(v)}(v'_1 \bowtie_e v'_2)$. The new graph $G(S')$ is obtained from $G(S)$ by removing the graph of v and adding the ones of v'_1 and v'_2.*

For example, cutting the join edge v_4:$n_1.s = n_2.s$ of $G(S_2)$ disconnects the graph of v_4, resulting in two new views, v_5 and v_6 (see Figure 3.1). View symbol v_4 is replaced in the rewritings by the expression $\pi_{head(v_4)}(v_5 \bowtie_{n_1.s=n_2.s} v_6)$. If we continue by cutting the edge v_3:$n_4.o = n_3.s$, v_3 is split into v_7 and v_8. The resulting state S_3 is:

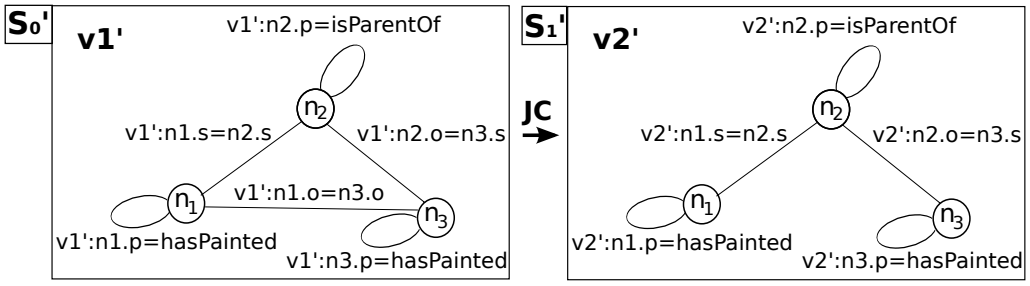


Figure 3.2: Initial state graph S'_0 and state S'_1 attained through a JC.

$$S_3 = \langle \{v_5, v_6, v_7, v_8\},$$
$$\{q_1 = \pi_{head(v_1)}(\pi_{head(v_2)}(\sigma_{n_1.o=starryNight}(\pi_{head(v_4)}(v_5 \bowtie_{n_1.s=n_2.s} v_6))) \bowtie$$
$$\pi_{head(v_3)}(v_7 \bowtie_{n_4.o=n_3.s} v_8)\}\rangle$$

The new views introduced in S_3 are the following:
$v_5(X, W)$:$-t(X, hasPainted, W)$
$v_6(X, Y)$:$-$ $t(X, isParentOf, Y)$
$v_7(X, Y)$:$-$ $t(X, isParentOf, Y)$
$v_8(Y, Z)$:$-$ $t(Y, hasPainted, Z)$

Since the above example covers only the second case of Definition 3.3.4, we will now provide an example that illustrates the first case as well. For this purpose, we use a slightly modified version of the sample query q_1, namely q'_1, in which we replaced the $starryNight$ constant of the first query atom with the variable Z (which also appears in the third atom). Query q'_1 asks for the painters that have worked on the same painting with one of their children, as well as these paintings, and is the following:

$q'_1(X, Z) :- t(X, hasPainted, Z), t(X, isParentOf, Y), t(Y, hasPainted, Z)$

The graph of the initial state S'_0 is depicted at the left-hand side of Figure 3.2. Performing a JC by cutting the join edge v'_1:$n_1.o = n_3.o$ of $G(S'_0)$ does not disconnect the graph of v'_1. A new state S'_1 is obtained, the graph of which is shown at the right-hand side of Figure 3.2. State S'_1 contains the following new view v'_2:

$v'_2(X, Z, W) :- t(X, hasPainted, W), t(X, isParentOf, Y), t(Y, hasPainted, Z)$

As dictated by the first case of the definition of JC, the first occurrence of the join variable Z in v'_1 is *replaced by a new fresh variable* W in v'_2. This is done so that we no longer have a join between the first and the third atom (if we kept the same variable name in both positions, the join would remain intact). The Z variable, still present in the third atom of v'_2, as well as the new fresh variable W, become head variables (Z already was in our example), so that in the rewritings that were using v'_1 we can re-impose (through a selection) the join that was removed by the JC. Thus, v'_1 is replaced in the rewritings by the expression $\pi_{head(v'_1)}(\sigma_{n1.o=n3.o} v'_2) = \pi_{X,Z}(\sigma_{W=Z} v'_2)$.

Definition 3.3.5 (View Fusion (VF)). *Let $S = \langle V, R\rangle$ be a state and v_1, v_2 be two views in V such that their respective graphs are isomorphic (their bodies are equivalent up to variable renaming). We denote by $\langle i \to j\rangle$ the renaming of the variables of v_i into those of*

v_j. *Let* v_3 *be a copy of* v_1*, such that* $head(v_3) = head(v_1) \cup head(v_{2\langle 2\to 1\rangle})$. *Fusing* v_1 *and* v_2 *leads to a new state* $S' = \langle V', R'\rangle$ *obtained as follows:*

- $V' = (V \setminus \{v_1, v_2\}) \cup \{v_3\}$,
- $G(S')$ *is obtained from* $G(S)$ *by removing the graphs of* v_1 *and* v_2 *and adding that of* v_3*, and*
- R' *is obtained from* R *by replacing any occurrence of* v_1 *with* $\pi_{head(v_1)}(v_3)$*, and of* v_2 *with* $\pi_{head(v_2)}(v_{3\langle 3\to 2\rangle})$

For example, in state S_3, the graphs of v_5 and v_8 are isomorphic, and can thus be fused creating the new view v_9. Similarly, v_6 and v_7 can be fused into a new view v_{10} leading to state S_4.

3.3.3 View and State Equivalence

As mentioned in Definition 3.3.5 above, in order to apply a VF between two views, we first need to check whether these views are equivalent (up to variable renaming). It is shown in [CM77] that in the general case, equivalence between conjunctive queries is NP-complete, which would make VF very expensive. To this end, we have devised a signature-based filter to more efficiently determine view equivalence. In particular, we assign a signature to each view taking into account the number of atoms of the view, the constants and the position (s, p or o) in which each constant appears, as well as the joins to which each variable participates (for each variable, we count the number of s-s, s-p, s-o, p-s, etc., joins to which it participates in the views). These signatures are small and can be built very fast.

If two views are equivalent, their signatures are the same. Thus, in order to test if $v_1 \equiv v_2$, we first compare their signatures. If they are different, we are sure the views are not identical. If they coincide, we apply the full equivalence test of [CM77]. The filter eliminates quickly many non-equivalent pairs and speeds up the whole process significantly.

As for the state equivalence, we build state signatures by sorting and then concatenating the signatures of all views composing the state. If two states have different signatures, they are certainly not equivalent. If they have the same signature, their views should be checked for equivalence, which is more costly. This efficient state equivalence test plays an important role when searching for candidate states, as will be explained in Section 3.5.1.

3.3.4 Estimated State Cost

To each state, we associate a *cost estimation* c^ϵ, taking into account: the space occupancy of all the materialized views, the cost of evaluating the workload query rewritings, and the cost associated to the maintenance of the materialized views.

For any conjunctive query or view v, we use $len(v)$ to denote the number of atoms in v, $|v|$ for the number of tuples in v, and $|v|^\epsilon$ for our *estimation* of this number. We now detail each of the three cost components. Let $S(Q) = \langle V, R\rangle$ be a state.

View space occupancy (VSO$^\epsilon$) To estimate the space occupancy of a given view $v \in V$, we need to estimate its cardinality. Several methods exist for estimating RDF query cardinality [MASS08, SSB$^+$08, NW09]. In this work, we adopt the solution of [NW09], which consists in counting and storing the exact number of tuples (i) for each given s, p and o value; (ii) for each pair of (s, p), (s, o) and (s, p) values. This leads to exact cardinality estimations for any one-atom view with one or two constants. The size of an one-atom view with no constants is the size of the dataset; three-constants atoms are disallowed in our framework since they introduce Cartesian products in views.

We now turn to the case of multi-atom views. From each view $v \in V$, and each atom $t_i \in v$, $1 \leq i \leq len(v)$, let v^i be the conjunctive query whose body consists of exactly the atom t_i and whose head projects the variables in t_i. From our gathered statistics, we know $|v^i|$. We assume that values in each triple table column are *uniformly distributed*, and that values of different columns are *independently distributed*[1]. For the s, p and o columns, moreover, we store the number of distinct values, as well as the minimum and maximum values. Then, we compute $|v|^\epsilon$ based on the exact counts $|v^i|$ and the above assumptions and statistics, applying known relational formulas [RG03]. Finally, to estimate the space occupancy of view v, we take into account $|v|^\epsilon$, along with the average size of a triple attribute (subject, property or object), and the number of attributes in the head of v.

Since the workload is known, we gather only the statistics needed for this workload. In particular, we count (i) the triples matching each of the query atoms, and (ii) the triples matching all *relaxations* of these atoms, obtained by removing constants (as SC does during the search). Consider, for instance, the following query:

$$q(X_1, X_2) :- t(X_1, rdf{:}type, picture), t(X_1, isLocatIn, X_2)$$

We count the triples matching the two query atoms:

$$q^1(X_1) :- t(X_1, rdf{:}type, picture), q^2(X_1, X_2) :- t(X_1, isLocatIn, X_2)$$

as well as the triples matching three *relaxed atoms*, obtained by removing the constants from q^1 and q^2:

$$q^3(X_1, X_2) :- t(X_1, rdf{:}type, X_2),\ q^4(X_1, X_2) :- t(X_1, X_2, picture),$$
$$q^5(X_1, X_2, X_3) :- t(X_1, X_2, X_3).$$

Based on the cardinalities of the above atoms, we can estimate the cardinality of any possible view created throughout the search.

Rewriting evaluation cost (REC$^\epsilon$) This cost estimation reflects the processing effort needed to answer the workload queries using the proposed rewritings in R. It is computed as:

$$REC^\epsilon(S) = \sum_{r \in R}(c_1 \cdot io^\epsilon(r) + c_2 \cdot cpu^\epsilon(r))$$

1. A recent work [NM11] provides an RDF query size estimation method that does not make the independence assumption. This estimation method could be also easily integrated in our framework.

where $io^\epsilon(r)$ and $cpu^\epsilon(r)$ estimate the I/O cost and the CPU processing cost of executing the rewriting r respectively, and c_1, c_2 are some weights. The I/O cost estimation is:

$$io^\epsilon(r) = \sum_{v \in r} |v|^\epsilon$$

where $v \in r$ denotes a view appearing in the rewriting r. The CPU cost estimation $cpu^\epsilon(r)$ sums up the estimated costs of the selections, projections, and joins required by the rewriting r, computed based on the view cardinality estimations and known formulas from the relational query processing literature [RG03].

View maintenance cost (VMC$^\epsilon$) The cost of maintaining the views in V when the data is updated, depends on the algorithm implemented to propagate the updates. In a conservative way, we chose to account only for the costs of writing/removing tuples to/from the views due to an update, ignoring the other maintenance operation costs. Consider the addition of a triple t_+ to the triple table, and a view v of $len(v)$ atoms. With some simplification, we consider that t_+ joins with f_1 existing triples for some constant f_1, the tuples resulting from this, in turn, join with f_2 existing triples, etc. Adding the triple t_+ thus causes the addition of $f_1 \cdot f_2 \cdot \ldots \cdot f_{len(v)}$ tuples to v. A similar reasoning holds for deletions. To avoid estimating $f_1, f_2, \ldots, f_{len(v)}$, which may be costly or impossible for triples which will be added in the future, we consider a single user-provided factor f, and compute:

$$VMC^\epsilon(S) = \sum_{v \in V} f^{len(v)}$$

The **estimated cost** c^ϵ of a state S is defined as:

$$c^\epsilon(S) = c_s \cdot VSO^\epsilon(S) + c_r \cdot REC^\epsilon(S) + c_m \cdot VMC^\epsilon(S)$$

where the numerical weights c_s, c_r and c_m determine the importance of each component: if storage space is cheap c_s can be set very low, if the triple table is rarely updated c_m can be reduced etc.

Workload query weights An immediate extension to the cost function is to associate numerical weights $W = \{w_1, w_2, \ldots, w_n\}$ to each workload query, to reflect, e.g., the frequency and/or importance of a query. To account for weights, the rewriting evaluation cost needs to become a weighted sum:

$$REC^\epsilon_W(S) = \sum_{r_i \in R} w_i \cdot (c_1 \cdot io^\epsilon(r_i) + c_2 \cdot cpu^\epsilon(r_i))$$

denoting by w_i the weight of the query q_i, having r_i as its corresponding rewriting. Weights have no other impact on our approach and will be omitted in the sequel for simplicity.

Impact of transitions on the cost Transition SC increases the view size and adds to some rewritings the CPU cost of the selection. Thus, SC always increases the state cost. Transitions JC and VB may increase or decrease the space occupancy, and add the costs of a join to some rewritings. JC decreases maintenance cost, whereas VB may increase or decrease it. Overall, JC and VB may increase or decrease the state cost. Finally, VF decreases the view space occupancy and view maintenance costs, and does not have an impact on the query processing cost. Thus, VF always reduces the overall cost of a state.

Semantic relationship	RDF notation	FOL notation
Class inclusion	$(c_1, \text{rdfs:subClassOf}, c_2)$	$\forall X(c_1(X) \Rightarrow c_2(X))$
Property inclusion	$(p_1, \text{rdfs:subPropertyOf}, p_2)$	$\forall X \forall Y(p_1(X,Y) \Rightarrow p_2(X,Y))$
Domain typing of a property	$(p, \text{rdfs:domain}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(X))$
Range typing of a property	$(p, \text{rdfs:range}, c)$	$\forall X \forall Y(p(X,Y) \Rightarrow c(Y))$

Table 3.1: Semantic relationships expressible in an RDFS.

3.4 View Selection & RDF Reasoning

The approach described so far does not take into consideration the implicit triples that are intrinsic to RDF and that complete query answers. Section 3.4.1 introduces the notion of RDF entailment to which such triples are due [2]. Section 3.4.2 presents the two main methods for processing RDF queries when RDF entailment is considered, namely *database saturation* and *query reformulation*. In particular, we devise a novel reformulation algorithm extending the state of the art. Finally, Section 3.4.3 details how we take RDF entailment into account in our view selection approach.

3.4.1 RDF entailment

The W3C RDF recommendation [www04a] provides a set of *entailment rules*, which lead to deriving new implicit (or *entailed*) triples from an RDF database. We provide here an overview of these rules.

Some implicit triples are obtained by generalizing existing triples using blank nodes. For instance, a triple (s, p, o) entails the triple $(\_{:}b, p, o)$, where s is a URI and $\_{:}b$ denotes a blank node.

Some other rules derive implicit triples from the semantics of a few special URIs, which are part of the RDF standard, and are assigned special meaning. For instance, RDF provides the rdfs:Class URI whose semantics is the set of all RDF-specific (predefined) *and* user-defined URIs denoting classes to which resources may belong. For example, when a triple states that a resource u belongs to a given user-defined class $painting$, i.e., $(u, \text{rdf:type}, painting)$ using the predefined URI rdf:type, an implicit triple states that $painting$ is a class: $(painting, \text{rdf:type}, \text{rdfs:Class})$.

Finally, some rules derive implicit triples from the semantics encapsulated in an *RDF Schema* (RDFS for short). An RDFS specifies semantic relationships between classes and properties used in descriptions. Table 3.1 shows the four semantic relationships allowed in RDF, together with their first-order logic semantics. Some rules derive implicit triples through the transitivity of class and property inclusions, and of inheritance of domain and range typing. For instance, if $painting$ is a subclass of $masterpiece$, i.e., $(painting, \text{rdfs:subClassOf}, masterpiece)$, which is a subclass of $work$, i.e., $(masterpiece, \text{rdfs:subClassOf}, work)$, then an entailed triple is

2. RDF entailment was also described in Chapter 2; here we recall its basic notions, and give more details on the part of entailment that is related to this work.

$(painting, \text{rdfs:subClassOf}, work)$. If $hasPainted$ is a subproperty of $hasCreated$, i.e., $(hasPainted, \text{rdfs:subPropertyOf}, hasCreated)$, the ranges of which are the classes $painting$ and $masterpiece$, respectively, i.e., $(hasPainted, \text{rdfs:range}, painting)$ and $(hasCreated, \text{rdfs:range}, masterpiece)$, then the following triples are implicit: $(hasPainted, \text{rdfs:range}, masterpiece)$, $(hasPainted, \text{rdfs:range}, work)$, and $(hasCreated, \text{rdfs:range}, work)$. Some other rules use the RDFS to derive implicit triples by propagating values (URIs, blank nodes, and literals) from subclasses and subproperties to their superclasses and superproperties, and from properties to classes typing their domains and ranges. If a resource u has painted something, i.e., $(u, hasPainted, \_{:}b)$, implicit triples are: $(u, hasCreated, \_{:}b)$, $(\_{:}b, \text{rdf:type}, painting)$, $(\_{:}b, \text{rdf:type}, masterpiece)$, and $(\_{:}b, \text{rdf:type}, work)$.

Returning complete answers requires considering all the implicit triples. In practice, RDF data management frameworks (e.g., Jena [3]) allow specifying the subset of RDF entailment rules w.r.t. which completeness is required. This is because the implicit triples brought by some rules, e.g., generalization of constants into blank nodes, may not be very informative in most settings. Of particular interest among all entailment rules are usually those derived from an RDFS, since they encode application domain semantics.

3.4.2 RDF entailment and query answering

We consider here the two main approaches previously proposed to answer queries w.r.t. a given set of RDF entailment rules: database saturation and query reformulation.

Database saturation The first approach *saturates* the database by adding to it all the implicit triples specified in the RDF recommendation [www04a]. The benefit of saturation is that standard query evaluation techniques for plain RDF can be applied on the resulting database to compute complete answers [www08]. Nevertheless, saturation also has drawbacks. First, it needs more space to store the implicit triples, competing with the data and the materialized views. Observe that saturation adds all implicit triples to the store, whether user queries need them or not. Second, the maintenance of a saturated database, which can be seen as an inflationary fixpoint, when adding or removing data and/or RDFS statements may be complex and costly. Finally, saturation is not always possible, e.g., when querying is performed at a client with no write access to the database.

Query reformulation The second approach *reformulates* a (conjunctive) query into an equivalent union of (conjunctive) queries. The complete answers of the initial query (w.r.t. the considered RDF entailment rules) can be obtained by standard query evaluation techniques for plain RDF [www08] using this union of queries against the non-saturated database.

The benefit of reformulation is that it leaves the database unchanged. However, reformulation has an overhead at query evaluation time.

Query reformulation w.r.t. an RDFS Query reformulation algorithms have been investigated in the literature for the well-known *Description Logic fragment* of RDF [AGR07, CGL$^+$07]: datasets complimented with an RDFS, without blank nodes, and where RDF

3. http://jena.sourceforge.net/

Algorithm 1: $\mathbf{Reformulate}(q, \mathcal{S})$

Input : an RDF schema $\mathcal{S}$ and a conjunctive query q over $\mathcal{S}$
Output: a union of conjunctive queries ucq such that for any database D:
$\text{evaluate}(q, \text{saturate}(D, \mathcal{S})) = \text{evaluate}(ucq, D)$

$ucq \leftarrow \{q\}$, $ucq' \leftarrow \emptyset$
while $ucq \neq ucq'$ **do**
  $ucq' \leftarrow ucq$
  foreach *conjunctive query* $q' \in ucq'$ **do**
    foreach *atom* g *in* q' **do**
      if $g = t(\mathtt{s}, rdf{:}type, c_2)$ *and* c_1 rdfs:subClassOf $c_2 \in \mathcal{S}$ **then**
        $ucq \leftarrow ucq \cup \{q'_{[g/t(\mathtt{s},rdf:type,c_1)]}\}$ //rule 3.1
      if $g = t(\mathtt{s}, p_2, \mathtt{o})$ *and* p_1 rdfs:subPropertyOf $p_2 \in \mathcal{S}$ **then**
        $ucq \leftarrow ucq \cup \{q'_{[g/t(\mathtt{s},p_1,\mathtt{o})]}\}$ //rule 3.2
      if $g = t(\mathtt{s}, rdf{:}type, c)$ *and* p rdfs:domain $c \in \mathcal{S}$ **then**
        $ucq \leftarrow ucq \cup \{q'_{[g/\exists X\ t(\mathtt{s},p,X)]}\}$ //rule 3.3
      if $g = t(\mathtt{o}, rdf{:}type, c)$ *and* p rdfs:range $c \in \mathcal{S}$ **then**
        $ucq \leftarrow ucq \cup \{q'_{[g/\exists X\ t(X,p,\mathtt{o})]}\}$ //rule 3.4
      if $g = t(\mathtt{s}, rdf{:}type, X)$ *and* $c_1, c_2 \ldots, c_n$ *are all the classes in* $\mathcal{S}$ **then**
        $ucq \leftarrow ucq \cup \bigcup_{i=1}^{n} \{(q'_{[g/t(\mathtt{s},rdf:type,c_i)]})_{\sigma=[X/c_i]}\}$ //rule 3.5
      if $g = t(\mathtt{s}, X, \mathtt{o})$ *and* $p_1, p_2 \ldots, p_m$ *are all the properties in* $\mathcal{S}$ **then**
        $ucq \leftarrow$ //rule 3.6
        $ucq \cup \bigcup_{i=1}^{m} \{(q'_{[g/t(\mathtt{s},p_i,\mathtt{o})]})_{\sigma=[X/p_i]}\} \cup \{(q'_{[g/t(\mathtt{s},rdf:type,\mathtt{o})]})_{\sigma=[X/rdf:type]}\}$
return ucq

entailment only considers the rules associated to the RDFS (those of the third kind described in Section 3.4.1). However, these algorithms allow reformulating queries from a strictly less expressive language than the one of our RDF queries (see Section 3.8 for more details) and, thus, cannot be applied to our setting. Therefore, we propose the Algorithm 1 that fully captures our query language, so that we can obtain the complete answers of any RDF query by evaluating its reformulation.

The algorithm uses the set of rules of Figure 3.3 to *unfold* the queries; in this Figure and onwards, we denote by $\mathtt{s}$, $\mathtt{p}$, respectively, $\mathtt{o}$, a placeholder for either a constant or a variable occurring in the subject, property, respectively, object position of a triple atom. Notice that rules (3.1)-(3.4) follow from the four rules of Table 3.1. The evaluate and saturate functions, used in Algorithm 1 provide, respectively, the standard query evaluation for plain RDF, and the saturation of a dataset w.r.t. an RDFS (Table 3.1). Moreover, $q_{[g/g']}$ is the result of replacing the atom g of the query q by the atom g' and $q_{\sigma=[X/c]}$ is the result of replacing any occurrence of the variable X in q with the constant c.

More precisely, Algorithm 1 uses the rules in Figure 3.3 to generate new queries from

$$t(\mathtt{s}, rdf{:}type, c_1) \Rightarrow t(\mathtt{s}, rdf{:}type, c_2), \text{with } c_1\ rdfs{:}subClassOf\ c_2 \in \mathcal{S} \quad (3.1)$$
$$t(\mathtt{s}, p_1, \circ) \Rightarrow t(\mathtt{s}, p_2, \circ), \text{with } p_1\ rdfs{:}subPropertyOf\ p_2 \in \mathcal{S} \quad (3.2)$$
$$t(\mathtt{s}, p, X) \Rightarrow t(\mathtt{s}, rdf{:}type, c), \text{with } p\ rdfs{:}domain\ c \in \mathcal{S} \quad (3.3)$$
$$t(X, p, \circ) \Rightarrow t(\circ, rdf{:}type, c), \text{with } p\ rdfs{:}range\ c \in \mathcal{S} \quad (3.4)$$
$$t(\mathtt{s}, rdf{:}type, c_i) \Rightarrow t(\mathtt{s}, rdf{:}type, X), \text{for any class } c_i \text{ of } \mathcal{S} \quad (3.5)$$
$$t(\mathtt{s}, p_i, \circ) \Rightarrow t(\mathtt{s}, X, \circ), \text{for any property } p_i \text{ of } \mathcal{S} \text{ and } rdf{:}type \quad (3.6)$$

Figure 3.3: Reformulation rules for an RDFS $\mathcal{S}$.

the original query, by a backward application of the rules on the query atoms. It then applies the same procedure on the newly obtained queries and repeats until no new queries can be constructed. Then, it outputs the union of the generated queries. The inner loop of the algorithm (lines 5-16) comprises six *if* statements, one for each of the six rules above. The conditions of these statements represent the heads (right parts) of the rules, whereas the consequents correspond to their bodies (left parts). In each iteration, when a query atom matches the condition of an *if* statement, the respective rule is triggered, replacing the atom with the one that appears in the body of the rule. Note that rules 3.5 and 3.6 need to bind a variable X of an atom to a constant c_i, p_i, or $rdf{:}type$, thus we use σ to bind all the occurrences of X in the query in order to retain the join on X within the whole new query.

We now prove the termination and correctness of $\mathbf{Reformulate}(q, \mathcal{S})$.

Theorem 3.4.1 (Termination of $\mathbf{Reformulate}(q, \mathcal{S})$)**.** *Given a query q over an RDFS $\mathcal{S}$, $\mathbf{Reformulate}(q, \mathcal{S})$ terminates and outputs a union of no more than $(2|\mathcal{S}|^2)^m$ queries, where $|\mathcal{S}|$ is the number of statements in $\mathcal{S}$ and m the number of atoms in q.*

Proof. For the algorithm to terminate, it suffices to show that rules (3.1)-(3.6) can be repeatedly applied a finite number of times to each atom of q. Let $|\mathcal{S}|$ be the number of statements in $\mathcal{S}$, $|R|$ be the number of relations (i.e., classes and properties), $|C|$ the number of classes and $|P|$ the number of properties participating in $\mathcal{S}$. Obviously, $|C| + |P| = |R|$. Observe that the atoms resulting from the application of rule 3.2 can only further trigger the same rule. Likewise, rules 3.3 and 3.4 can only trigger rule 3.2, whereas rule 3.1 only enables the application of rules 3.1, 3.3, and 3.4. Rule 3.5 only triggers rules 3.1, 3.3, and 3.4. Lastly, rule 3.6 may trigger any other rule.

Clearly, the worst case (longest sequence of rule applications) occurs when rule 3.6 is applied on an atom of the form $t(\mathtt{s}, X, Y)$: it generates $|P|$ atoms (as many as the distinct properties of $\mathcal{S}$) of the form $t(\mathtt{s}, p_j, Y)$ and one atom of the form $t(\mathtt{s}, rdf{:}type, Y)$. Each of the $|P|$ atoms can cause the recursive application of rule 3.2, as explained before. Rule 3.2 can be applied at most $|\mathcal{S}|$ times for each of these atoms (in case $\mathcal{S}$ includes only $\mathrm{subPropertyOf}$ statements), leading to a total number of $|P||\mathcal{S}|$ generated atoms. As for the atom $t(\mathtt{s}, rdf{:}type, Y)$ also output by rule 3.6, it can trigger rule 3.5, which then generates $|C|$ new atoms (as many as the distinct classes in $\mathcal{S}$) of the form $t(\mathtt{s}, rdf{:}type, c_i)$. As explained above, each of these $|C|$ atoms can enable the recursive application of rules 3.1,

3.3 and 3.4 at most $|\mathcal{S}|$ times, leading to $|C||\mathcal{S}|$ atoms. Summing up, we can have at most $|P||\mathcal{S}| + |C||\mathcal{S}| = (|P| + |C|)|\mathcal{S}| = |R||\mathcal{S}|$ rule applications. Now observe that the biggest number of distinct relations that can appear in $\mathcal{S}$ is $2|\mathcal{S}|$, which happens when no relation is used more than once in the statements in $\mathcal{S}$. Thus, the above sum is updated to $2|\mathcal{S}||\mathcal{S}| = 2|\mathcal{S}|^2$ which constitutes the upper bound for the number of reformulations per atom. That is, if q comprises m atoms, $\mathbf{Reformulate}(q, \mathcal{S})$ terminates after at most $(2|\mathcal{S}|^2)^m$ rule applications, leading to an equal number of queries. □

Theorem 3.4.2 (Correctness of Algorithm 1)**.** *Let ucq be the output of $\mathbf{Reformulate}(q, \mathcal{S})$, for a query q over an RDFS $\mathcal{S}$. For any database D associated to $\mathcal{S}$:*

$$\text{evaluate}(q, \text{saturate}(D, \mathcal{S})) = \text{evaluate}(ucq, D).$$

Proof (Sketch). **Soundness** We show that if tuple $t \in \text{evaluate}(ucq, D)$, then $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$. Since $t \in \text{evaluate}(ucq, D)$, t is an answer to a query q' in ucq. Moreover, since $D \subseteq \text{saturate}(D, \mathcal{S})$, we have $\text{evaluate}(q', D) \subseteq \text{evaluate}(q', \text{saturate}(D, \mathcal{S}))$. By construction, any query built by Algorithm 1 is subsumed by q w.r.t. $\mathcal{S}$, so $\text{evaluate}(q', \text{saturate}(D, \mathcal{S})) \subseteq \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, thus $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$.

Completeness We now show that if tuple $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, then $t \in \text{evaluate}(ucq, D)$. Since $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, t results from a projection upon m triples $t_1, \ldots, t_m$, given that q has m atoms. We therefore have to show that any t_i is also exhibited by a reformulation of the i-th atom of q.

First, observe that our set of rules is capable of capturing all possible cases of query atoms. Clearly, every atom $t(s, p, o)$ consists of three terms, each being either a variable or a constant. When an atom contains a variable in p, rule (3.6) is triggered, whereas when p is specified, rules (3.2) or (3.5) can be used (depending on whether p is $rdf{:}type$ or not, respectively). Finally, when p is $rdf{:}type$ and o is some constant, rules (3.1), (3.3) and (3.4) can be applied. Hence, all cases of query atoms are treated.

Now, we prove the above claim by induction on the number α of applications of the saturation rules, needed for t_i to be added in $\text{saturate}(D, \mathcal{S})$. These rules are the ones of Table 3.1, applied in a forward-chaining fashion [www04a]. We actually show that the free variables of the i-th atom of q that are bound by t_i, are equally bound by the evaluation of a reformulation of the i-th atom of q. For $\alpha = 0$, $t_i \in D$ (i.e., t_i is an explicit triple), thus t_i is also a triple for the evaluation of the non-reformulated i-th atom of q. Suppose that the claim holds for $\alpha < k$, and let us consider the case for $\alpha = k$. Assume t_i is finally added after the application of the first closure rule on a triple $t_{\alpha-1}$. Then, $t_{\alpha-1} = (s_1, rdf{:}type, c_1)$ and $t_i = (s_1, rdf{:}type, c_2)$, where s_1, c_1 and c_2 are constants. Since $t \in \text{evaluate}(q, \text{saturate}(D, \mathcal{S}))$, t_i matches the i-th atom of q, which is, thus, of the form $t(\mathtt{s}, rdf{:}type, c_2)$, $t(\mathtt{s}, X, c_2)$, $t(\mathtt{s}, rdf{:}type, X)$, or $t(\mathtt{s}, X, Y)$.

In the first case, we perform a reformulation of the i-th atom using rule 3.1 and we obtain the atom $t(\mathtt{s}, rdf{:}type, c_1)$, which indeed returns s_1 in the result, as if the triple t_i was stored in D. In the second case, we reformulate the query atom with rule 3.6 and we obtain (among others) the atom $t(\mathtt{s}, rdf{:}type, c_2)$, which, after one more reformulation using rule 3.1, results in the atom $t(\mathtt{s}, rdf{:}type, c_1)$ that was treated by the first case. In

the third case, we apply rule 3.5, we immediately obtain the atom $t(\mathtt{s}, rdf{:}type, c_1)$ and so we also return s_1 in the result. In the last case, we apply rule 3.6 and on the new atom $t(\mathtt{s}, rdf{:}type, X)$ we apply rule 3.5 and then fall into the previous case.

Thus, for all four cases that can appear after applying the first saturation rule, we have proved by induction our claim. We proceed the same way for the three other saturation rules. □

3.4.3 View selection aware of RDF entailment

We now discuss possible ways to take RDF implicit triples into account in our view selection approach. As will be explained, the exact way (cardinality) statistics are collected for each view atom (first described in Section 3.3.4), plays an important role here.

Database saturation If the database is saturated prior to view selection, the collected statistics do reflect the implicit triples.

Pre-reformulation Alternatively, one could reformulate the query workload and then apply our search on the new workload. To do so, we extend the definition of our initial state, as well as our rewriting language to that of *unions* of conjunctive queries. More precisely, given a set of queries $Q = \{q_1, \ldots, q_n\}$, and assuming that $\mathrm{Reformulate}(q_i, \mathcal{S}) = \{q_i^1, \ldots, q_i^{n_i}\}$, it is sufficient to define $S_0(Q) = \langle V_0, R_0 \rangle$ as the set of conjunctive views $V_0 = \bigcup_{i=1}^{n} \{q_i^1, \ldots, q_i^{n_i}\}$ and the set of rewritings $R_0 = \bigcup_{i=1}^{n} \{q_i = q_i^1 \cup \cdots \cup q_i^{n_i}\}$. In this case, statistics are collected on the original (non-saturated) database for the reformulated queries.

As stated in Theorem 3.4.1, query reformulation can yield a significant number of new queries, increasing the number of views of our initial state and leading to a serious increase of the search space. As an example, consider the following simple query on the Barton [www] dataset:

$$q(X_1, X_2, X_3) {:}\!- t(X_1, rdf{:}type, text), t(X_1, relatedTo, X_2),\\ t(X_2, rdf{:}type, subjectPart), t(X_1, language, fr),\\ t(X_2, description, X_3)$$

q is reformulated with the Barton Schema into a union of 104 queries. Given the very high complexity of the exhaustive search problem (Section 3.5.1), such an increase may significantly impact view selection performance.

Post-reformulation To avoid this explosion, we propose to apply reformulation not on the initial queries, but directly on the views in the final (best) state recommended by the search.

Directly doing so, introduces a source of errors: since statistics are collected on the original database, and the queries are not reformulated, the implicit triples will not be taken into account in the cost estimation function c^ϵ. To overcome this problem, we reflect implicit triples *to the statistics, by reformulating each view atom v^i into a union of atoms* $\mathrm{Reformulate}(v^i, \mathcal{S})$ *prior to the view search, and then replacing* $|v^i|$ *(i.e., the cardinality of v^i) in our cost formulas with* $|\mathrm{Reformulate}(v^i, \mathcal{S})|$. This results in having the same statistics as if the database was saturated. Then, we perform the search using the (non-reformulated) queries and get the same best state as in the database saturation approach (as we use the same initial state and statistics). Since materializing the best

$q^{1,\mathcal{S}}$	$q^1(X_1)$	$:-t(X_1, rdf{:}type, picture)$	(1)
	$\cup\, q^1(X_1)$	$:-t(X_1, rdf{:}type, painting)$	(2)
$q^{4,\mathcal{S}}$	$q^4(X_1, X_2)$	$:-(X_1, X_2, picture)$	(1)
	$\cup\, q^4(X_1, isLocatIn)$	$:-t(X_1, isLocatIn, picture)$	(2)
	$\cup\, q^4(X_1, isExpIn)$	$:-t(X_1, isExpIn, picture)$	(3)
	$\cup\, q^4(X_1, rdf{:}type)$	$:-t(X_1, rdf{:}type, picture)$	(4)
	$\cup\, q^4(X_1, isLocatIn)$	$:-t(X_1, isExpIn, picture)$	(5)
	$\cup\, q^4(X_1, rdf{:}type)$	$:-t(X_1, rdf{:}type, painting)$	(6)

Table 3.2: Term reformulation for post-reasoning.

state's views directly would not include the implicit triples, we need to reformulate these views first. Theorem 3.4.2 guarantees the correctness of post-reformulation (materializing the reformulated views on the non-saturated database is the same as materializing the non-reformulated ones on the saturated database).

Consider the query q of Section 3.3.4, with the following Schema:

$$\mathcal{S} = \{(painting, \text{rdfs:subClassOf}, picture),\\ (isExpIn, \text{rdfs:subPropertyOf}, isLocatIn)\}$$

We first count the exact number of triples matching the query atoms and their relaxed versions, namely q^1 to q^5 (see Section 3.3.4).

We now reformulate each q^i based on $\mathcal{S}$ into a union of queries, denoted $q^{i,\mathcal{S}}$. Table 3.2 illustrates this for q^1 and q^4. Rule 1 (Figure 3.3) has been applied on q^1, adding to it a second union term. Applying rule 6 on q^4 leads to replacing X_2 with $isLocatIn$, $isExpIn$, and $rdf{:}type$ respectively in the second, third and fourth union terms of $q^{4,\mathcal{S}}$. In turn, the second term triggers rule 2 producing a fifth term, while the fourth term triggers rule 1 to produce the sixth union term.

The cardinality of each reformulated atom $q^{i,\mathcal{S}}$ is estimated prior to the search. Then, we perform the search for the non-reformulated version of q using these statistics, and get the following best state:

$$v_1(X_1, X_2){:-}t(X_1, rdf{:}type, X_2),\ v_2(X_1, X_2){:-}t(X_1, isLocatIn, X_2)$$
$$r_3 = \pi_{v_1.X_1, v_2.X_2}(\sigma_{X_2=picture}(v_1) \bowtie_{v_1.X_1=v_2.X_1} v_2)$$

After the search has finished, instead of the recommended views v_1 and v_2, we materialize their reformulated variants v_1' and v_2':

$$\begin{gathered} v_1'(X_1, X_2){:-}t(X_1, rdf{:}type, X_2)\\ \cup\, v_1'(X_1, painting){:-}t(X_1, rdf{:}type, painting)\\ \cup\, v_1'(X_1, picture){:-}t(X_1, rdf{:}type, picture)\\ \cup\, v_1'(X_1, picture){:-}t(X_1, rdf{:}type, painting) \end{gathered}$$

$$\begin{gathered} v_2'(X_1, X_2){:-}t(X_1, isLocatIn, X_2)\\ \cup\, v_2'(X_1, X_2){:-}t(X_1, isExpIn, X_2) \end{gathered}$$

Executing r_3 on v_1' and v_2' provides the complete answers for q.

In post-reformulation, finding the best state does not require saturating the database, nor multiplying the queries (as pre-reformulation does) and making the search space size

explode. Thus, this is the best approach for situations where database saturation is not an option, which is also shown through our experiments in Section 3.7.5.

Some measurements about the cost of statistics collection for each of the three reasoning approaches are also given in Section 3.7.5.

3.5 Searching for View Sets

This Section discusses strategies for navigating in the search space of candidate view sets (or states), looking for a low- or minimal-cost state. We discuss the exhaustive search strategies and identify an interesting subset of *stratified* strategies in Section 3.5.1, based on which we analyze the size of the search space. In Section 3.5.2, we present several efficient optimizations and search heuristics.

3.5.1 Exhaustive Search Strategies

We define the *initial state* of the search as $S_0(Q) = \langle V_0, R_0 \rangle$, such that $V_0 = Q$, i.e., the set of views is exactly the set of queries, and each rewriting in R_0 is a view scan. The state graph $G(S_0)$ corresponds to the queries in Q. Clearly, the rewriting cost of S_0 is low, since each query rewriting is simply a view scan. However, its space consumption and/or view maintenance costs may be high.

We denote by $S \xrightarrow{\tau} S'$ the application of the transition $\tau \in \{\text{SC},\text{JC},\text{VB},\text{VF}\}$ on a state S, leading to the state S'.

Definition 3.5.1 (Path)**.** *A path is a sequence of transitions of the form:* $S_0 \xrightarrow{\tau_0} S_1$, $S_1 \xrightarrow{\tau_1} S_2$, …, $S_{k-1} \xrightarrow{\tau_{k-1}} S_k$.

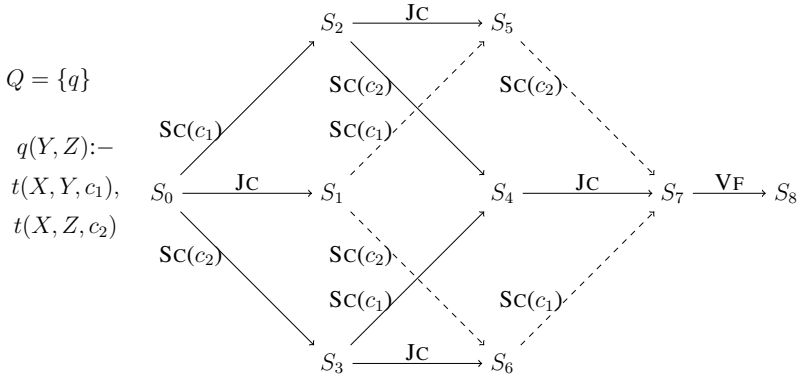
For instance, in Figure 3.4, $(S_0 \xrightarrow{\text{SC}(c_2)} S_3), (S_3 \xrightarrow{\text{JC}} S_6)$ is a path. We may denote a path simply by its transitions, e.g., (SC(c_2), JC).

It can be shown that any path is *cycle-free*. The intuition is that SC and JC remove query-specified predicates from the views, and no transition ever brings them back. Similarly, VB and JC always create smaller views, while no transition replaces a view (or two views) by a larger one. It follows that any path is of finite length.

Theorem 3.5.1 (Completeness of the transition set)**.** *Given a workload Q and an initial state S_0, for every possible state $S(Q)$, there exists a path from the initial state S_0 to S.*

Proof. Given a workload Q, an initial state S_0 and a possible state $S(Q) = \langle V, R \rangle$ that corresponds to a candidate view set for Q, we show that S can be attained through a sequence of transitions, all of which belong to our transition set $\{\text{SC},\text{JC},\text{VB},\text{VF}\}$.

By the definition of the candidate view set, we know that from every view v, there exists an embedding ϕ into at least one query $q_v \in Q$, otherwise, v would not be usable in any query rewriting. If ϕ mapped two atoms of v to the same atom of q_v, then v would be non-minimal, which contradicts our definition of the candidate view set (see Definition 3.2.3). Thus, ϕ maps each v atom to a distinct q_v atom, which entails that v has at most as many atoms as q_v.



V_0 $\{q(Y,Z){:-}t(X,Y,c_1), t(X,Z,c_2)\}$
V_1 $\{q_1(X_1,Y){:-}t(X_1,Y,c_1);\quad q_2(X_2,Z){:-}t(X_2,Z,c_2)\}$
V_2 $\{q(Y,Z,W_1){:-}t(X,Y,W_1), t(X,Z,c_2)\}$
V_3 $\{q(Y,Z,W_2){:-}t(X,Y,c_1), t(X,Z,W_2)\}$
V_4 $\{q(Y,Z,W_1,W_2){:-}t(X,Y,W_1), t(X,Z,W_2)\}$
V_5 $\{q_1(X_1,Z,W_1){:-}t(X_1,Z,W_1);\quad q_2(X_2,Z){:-}t(X_2,Z,c_2)\}$
V_6 $\{q_1(X_1,Z){:-}t(X_1,Z,c_1);\quad q_2(X_2,Z,W_2){:-}t(X_2,Z,W_2)\}$
V_7 $\{q_1(X_1,Z,W_1){:-}t(X_1,Z,W_1);\quad q_2(X_2,Z,W_2){:-}t(X_2,Z,W_2)\}$
V_8 $\{q(X,Y,Z){:-}t(X,Y,Z)\}$

Figure 3.4: Sample exhaustive strategy (solid arrows), EXNAÏVE strategy (solid and dashed arrows), and view sets corresponding to each state.

To start with, assume that our workload Q consists of a single query q. Each state consists of a possible view set that can be used to rewrite q. We initially assume that no rewriting uses a specific view more than once. We now distinguish two cases depending on the number of views $|V|$ in S:

$|\mathbf{V}| = \mathbf{1}$. In this case q is rewritten based only on one view v. Then v must have the same number of atoms as q: we have already shown that v cannot have more atoms than q; if v had less atoms than q, it could not suffice to rewrite q. Moreover, the graph of v has to be a subgraph of q (i.e., less restrictive than q) in order to be able to answer q. By repeatedly applying a number of SC and JC starting from q, we can obtain all possible subgraphs of q, among which we find the graph of the given view v. This means that in this case we can reach S starting from S_0.

$|\mathbf{V}| \geq \mathbf{1}$. Now assume that V contains $k > 1$ views, all of which participate to the (single) rewriting r of q. Hence, all views can be embedded into q. Notice though, that for every two views $v_1, v_2 \in V$, the nodes of q to which v_1 is mapped cannot be a subset of the nodes of q to which v_2 is mapped, because this entails the existence of a rewriting r' which does not use v_1, making r non-minimal, which contradicts our assumptions [4] (Definition 3.2.3). Starting from the initial view $q \in S_0$, we can use a sequence of JC and VB to split q into a set of views V_q, containing exactly k views. The k views of V_q are created so as to establish a one-to-one correspondence between V and V_q, as follows. For each view $v \in V$, we create exactly one view $v_1 \in V_q$ consisting of the q atoms to

4. This is also the reason we impose the restriction in the definition of VB that when a view is broken into two new views with node sets N_1 and N_2, we should have $N_1 \not\subseteq N_2$ and $N_2 \not\subseteq N_1$.

which v is mapped. This means that by construction v can be embedded into v_q. Thus, the graph of v is a subgraph of v_q and since they have the same number of nodes (as v and v_q have the same number of atoms), we can reach v from v_q by a sequence of SC and JC, as explained in the previous case when $|V| = 1$. Thus, starting from q we have shown how we can create the set V and reach from S_0 the state S.

We now turn to the case when we still have one query q in our workload, but the rewriting of q can use the same view more than once. We construct a new state S' in which we have created for each view of the view set V of S as many copies as the number of times it is used in the rewriting of q. Thus, in S' the rewriting of q includes views that are used only once. Reaching S from S' can be done easily by applying a sequence of VFs which revert the view duplication steps that brought us to S'. Moreover, showing that S' is attainable from S_0 falls into the case when $|V| \geq 1$ that was described above. Hence, S can be indeed attained from S_0.

Assume now that we have more than one, say m, queries in Q. The set of rewritings R of the given state S contains m rewritings. We distinguish two cases:

No view in V is used in more than one rewriting. In this case, we can treat each query $q \in Q$ as a separate initial state and, as explained above, construct all the views that participate in the rewriting of q. This way we will obtain m new states, one for every query. Once this is done, we combine the m states and create a single state that has as views the union of views of the individual "one-query" states, and as rewritings the union of rewritings of the m states. This is equivalent to the state S.

Some views in V are used in multiple rewritings. Given a state S, we construct a new state S' in which we copy each V view that participated in n_v rewritings in S, into n_v distinct views, each of which participates to exactly one rewriting in S' [5]. It follows from the case above (when no view could be used in more than one rewriting) that S' can be reached from S_0 using our set of transitions. Moreover, one can easily reach S from S' by simply applying a sequence of VFs which revert the view duplication steps we took to obtain S'. Thus, we have shown that S can be obtained from S_0. □

Definition 3.5.2 (Strategy)**.** *A search strategy Σ is a sequence of transitions of the form:*

$$\Sigma = (S_{i_1} \xrightarrow{\tau_{i_1}} S'_{i_1}), (S_{i_2} \xrightarrow{\tau_{i_2}} S'_{i_2}), \ldots, (S_{i_{k-1}} \xrightarrow{\tau_{i_{k-1}}} S'_{i_{k-1}}), (S_{i_k} \xrightarrow{\tau_{i_k}} S'_{i_k})$$

where $S_{i_1} = S_0$, for every $j \in [1..k]$ $\tau_{i_j} \in \{\text{SC}, \text{JC}, \text{VB}, \text{VF}\}$, and for every $j \in [2..k]$ there exists $l < j$ such that $S'_{i_l} = S_{i_j}$ (each state but S_0 must be attained before it is transformed).

For example, for the one-query workload depicted at the top left of Figure 3.4, one possible strategy is:

$$\Sigma_1 = (S_0 \xrightarrow{\text{SC}(c_1)} S_2), (S_2 \xrightarrow{\text{SC}(c_2)} S_4), (S_0 \xrightarrow{\text{SC}(c_2)} S_3),$$
$$(S_3 \xrightarrow{\text{SC}(c_1)} S_4), (S_0 \xrightarrow{\text{JC}} S_1)$$

5. Observe that our definitions of views and candidate view sets do not preclude the existence of two identical views in the same candidate view set, as long as each view is minimal.

Algorithm 2: EXNAÏVE(S_0)

Input : an initial state S_0
Output: the best state S_b found

```
1 S_b ← S_0, S_new ← null, CS ← {S_0}, ES ← ∅
2 while CS ≠ ∅ do
3     foreach state S_c ∈ CS do
4         S_new ← applyTrans({SC,JC,VB,VF}, S_c, (ES ∪ CS))
5         if S_new = null then move S_c from CS to ES
6         else
7             CS ← CS ∪ {S_new}
8             if c^ε(S_new) < c^ε(S_b) then S_b ← S_new
```

A strategy Σ is *exhaustive* if any state S that can be reached through a path, is also reached in Σ (not necessarily through the same path). For instance, in Figure 3.4, the solid arrows depict an exhaustive strategy, reaching all possible states.

We first consider a simple family of strategies called EXNAÏVE and described through Algorithm 2. EXNAÏVE strategy (as all strategies presented in this work) maintains a *candidate state set* CS and a set of *explored states* ES. CS keeps the states on which more transitions can be possibly applied and is initially $\{S_0\}$. ES is disjoint from CS and is empty in the beginning. A state S is explored, when any state $S' = \tau(S)$ obtained by applying some transition $\tau \in \{\text{SC, JC, VB, VF}\}$ to S, already belongs either to CS or to ES. EXNAÏVE at each point picks a state S_c from CS and tries to apply a transition to it ($applyTrans$, line 4). If no new state is obtained, S_c was already explored and is moved to ES (line 5); otherwise, the newly obtained state (S_{new}) is copied to CS (line 7). During the search, we also keep the *best state* found so far (denoted S_b), i.e., having the lowest cost $c^\epsilon(S)$ (line 8). The strategy stops when no new states can be found. Clearly, EXNAÏVE strategies are exhaustive. In Figure 3.4, the solid and dashed arrows, together, illustrate an EXNAÏVE strategy.

Note that checking whether a state S belongs to CS or ES is done using the signature-based filter presented in Section 3.3.3. To this end, we organized the CS and ES sets as hash maps where on a state signature we keep one or several states. This speeds up the process of look-up, since we search in CS and ES directly by a look-up on S's signature.

For a given strategy Σ, the *paths to a state* $S \in \Sigma$, denoted $^{\hookrightarrow}S$, is the set of all Σ paths whose final state is S. In an EXNAÏVE strategy there may be multiple paths to some states, e.g., S_6 is reached twice in our example, which slows down the search. We define the notion of *stratification* to reduce the number of such duplicate states.

Definition 3.5.3 (Stratified path)**.** *A path $p \in {}^{\hookrightarrow}S$ for some state $S \in \Sigma$ is stratified iff it belongs to the regular language:* VB* SC* JC* VF*.

A stratified path constrains the order among the types of transitions on the path: all possible view breaks appear only in the beginning of the path and are followed by the

selection cuts. Join cuts appear only after all selection cuts are applied and are in turn followed by zero or more view fusions. In Figure 3.4, all solid-arrow paths starting from S_0 are stratified.

The following theorem formalizes the interest of stratified paths.

Theorem 3.5.2 (Completeness of stratified paths)**.** *Let Q be a query workload and $S(Q)$ be a state for Q. There exists a stratified path leading from the initial state S_0 to S.*

Proof. If S is a state for Q, due to Theorem 3.5.1, there exists a path p belonging to some strategy Σ (for instance, an ExNaïve strategy) reaching S. We show that if p is not stratified, it can be transformed into a stratified path p', which also has S as its final state. Notice that paths including a single transition are always stratified, so hereafter, we focus only on paths of bigger length.

For a given transition τ appearing in a path p, we define the *forward inversion count of τ in p* (denoted $FIC(\tau, p)$) as the number of transitions that appear *after* τ in p, and that should appear *before* τ if p was stratified. For instance, consider the path $p_1 =$ (SC$_1$, VF$_1$, JC$_1$, VF$_2$, SC$_2$, VB$_1$, JC$_2$) where the states are not shown, and transitions of the same kind are distinguished by their subscripts. We have $FIC(\text{SC}_2, p_1) = 1$, since VB$_1$ appears after SC$_2$; the other transition appearing after SC$_2$, namely JC$_2$, does not violate stratification. Similarly, $FIC(\text{JC}_1, p_1) = 2$ since SC$_2$ and VB$_1$ appear after JC$_1$ in p_1; moreover, $FIC(\text{VB}_1, p_1) = 0$ and $FIC(\text{VF}_2, p_1) = 3$. Clearly, for any $\tau \in p$, $0 \leq FIC(\tau, p) < |p|$, where $|p|$ is the number of transitions in p.

Further, we define the forward inversion count of a path p as the sum of all the FICs of the transitions in p, that is: $FIC(p) = \sum_{\tau \in p} FIC(\tau, p)$. Path p is stratified if and only if $FIC(p) = 0$.

We now turn to consider our non-stratified path p. From the definition of stratified paths, one easily derives a set of six elementary stratification violations: these are path fragments of the form (τ_1, τ_2), each of which contradicts stratification, and at least one of which must be present in any non-stratified path. For each such violating fragment p_v going from a state S_1 to another state S_3, we provide another path fragment p_s going from S_1 to S_3, and which is stratified.

Case 1 ($p_v = S_1 \xrightarrow{\text{JC}} S_2 \xrightarrow{\text{SC}} S_3$). By the semantics of SC and JC, it follows readily that, since SC and JC target different edges of the state graph, there exists a state S_1', such that we can now reach S_3 through the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{JC}} S_3$.

Case 2 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{SC}} S_3$). Here we distinguish two cases:

- If VF and SC are performed at different places of S_1 (VF fuses two views, while SC cuts an edge from a third, distinct view), then we can apply them in the inverse order and reach S_3 with the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{VF}} S_3$. This case is similar to Case 1 above.
- In p_v, when SC erases a selection on a constant appearing in the fused view resulting from VF , we need to apply two SC steps prior to VF in order to attain S_3 from S_1 through the stratified path $p_s = S_1 \xrightarrow{\text{SC}} S_1' \xrightarrow{\text{SC}} S_2' \xrightarrow{\text{VF}} S_3$.

Case 3 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{JC}} S_3$). Three sub-cases can occur:

- When VF and JC are applied on different views, we can reach reach S_3 through the stratified $p_s = S_1 \xrightarrow{\text{JC}} S'_1 \xrightarrow{\text{VF}} S_3$.
- When JC is applied on the view resulting from VF and it does not disconnect this view, we need the stratified path $p_s = S_1 \xrightarrow{\text{JC}} S'_1 \xrightarrow{\text{JC}} S'_2 \xrightarrow{\text{VF}} S_3$, whereas
- if it disconnects the view, we need two JC steps and then two VF, that is the path $p_s = S_1 \xrightarrow{\text{JC}} S'_1 \xrightarrow{\text{JC}} S'_2 \xrightarrow{\text{VF}} S'_3 \xrightarrow{\text{VF}} S_3$.

Case 4 ($p_v = S_1 \xrightarrow{\text{SC}} S_2 \xrightarrow{\text{VB}} S_3$). In a way similar to the above cases, if SC and VB affect different views, it suffices to use the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{SC}} S_3$, while when VB is performed on the view resulting from SC, we need the new path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{SC}} S'_2 \xrightarrow{\text{SC}} S_3$.

Case 5 ($p_v = S_1 \xrightarrow{\text{JC}} S_2 \xrightarrow{\text{VB}} S_3$). In the simple case that JC and VB are applied on different views, we can reach S_3 through the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{JC}} S_3$. When JC and VB affect the same view, we distinguish the following sub-cases:

- Assume that JC does not disconnect the view on which it is applied. Notice that, by definition, VB may remove some of the edges of the view to which it is applied. If the edge that JC removes would also be removed by VB (if it is applied prior to JC), then we can omit JC and reach S_3 through the path $p_s = S_1 \xrightarrow{\text{VB}} S_3$. In the opposite case, JC removed an edge that has to be also removed from both the views output by VB. Thus, we need the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{JC}} S'_2 \xrightarrow{\text{JC}} S_3$.
- If JC disconnected the view, then the sequence of JC followed by VB created lead to three views, and JC was applied in a different part of the initial view. Thus, we can simply use the inverse order of transitions and still reach S_3, through the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{JC}} S_3$.

Case 6 ($p_v = S_1 \xrightarrow{\text{VF}} S_2 \xrightarrow{\text{VB}} S_3$). In this case, if VF and VB affect different views, we need the stratified path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{VF}} S_3$, while when VB is performed on the view resulting from VF, we need the new path $p_s = S_1 \xrightarrow{\text{VB}} S'_1 \xrightarrow{\text{VB}} S'_2 \xrightarrow{\text{VF}} S'_3 \xrightarrow{\text{VF}} S_3$.

Applying one of the path substitutions above on a sub-path $p_v = (\tau_1, \tau_2)$ of p turns it into a new path p', in which p_v is replaced with a subpath p_s of one of the following forms: (τ'_2, τ'_1), $(\tau'_2, \tau'_1, \tau''_1)$, $(\tau'_2, \tau''_2, \tau'_1)$ or $(\tau'_2, \tau''_2, \tau'_1, \tau''_1)$ [6]. In all cases, we have:

$$FIC(\tau'_1, p') = FIC(\tau''_1, p') = FIC(\tau_1, p) - 1$$
$$FIC(\tau'_2, p') = FIC(\tau''_2, p') = FIC(\tau_2, p)$$

In other words, the FIC of the first among the two transitions is diminished by 1. This is because (i) the path substitution ensures that τ'_2 (and τ''_2) no longer contributes to the FIC of τ'_1 (and τ''_1), (ii) no other transition introduced by the path substitution contributes to $FIC(\tau_1)$ and (iii) the FICs of p transitions after the end of p_v are unaffected by the substitution.

To turn p into a stratified path p', our first step is to bring all VFs at the end of the path. To do so, we repeatedly identify the last VF transition in the current path p_c, call it VF_k, for which $FIC(\text{VF}_k, p_c) \neq 0$ and apply on VF_k and its successor transition, the

6. Notice that τ'_1, τ''_1 are of the same kind as τ_1, and τ'_2, τ''_2 are of the same kind as τ_2.

path substitution which is appropriate (one of Cases 2, 3 and 6 is sure to apply). Each step reduces $FIC(\text{VF}_k, p_c)$ by 1. At the end of this step, the FIC of all VF transitions in the transformed path is 0, which also means that all VFs are placed at the end of the path.

We now continue the procedure for the JCs: we identify the last occurrence of JC, call it JC_k, for which $FIC(\text{JC}_k, p_c) \neq 0$ and we apply Case 1 or 5, depending on its successor transition. When this step is finished, we repeat the procedure for SCs to place them immediately before the JCs. We do not need to do the same for VBs, as previous operations have already pushed them at the beginning of the resulting path p' which, by now, is stratified, i.e., for all its transitions τ we will have $FIC(\tau, p') = 0$ and, thus, $FIC(p') = 0$.

Notice though that some path substitutions increase the length of the path, e.g., (SC,VB) may be replaced by (VB,SC,SC). However, for all violations there exists one case that does not affect the length of the path. At the same time, a bound holds on the maximal path size, as follows:

- The number of VBs that can be applied successively on a given state is limited, because each time the views that result from a VB have less atoms than the view VB is applied on.
- SCs and JCs are bound by the number of edges in the graphs of the views, which is finite.
- An infinite number of VFs would mean an infinite number of initial queries, which is a contradiction.

Thus, the above procedure always terminates and results in a stratified path p'. □

We can now identify an interesting family of strategies.

Definition 3.5.4 (Stratified strategy)**.** *A strategy Σ is stratified iff for any $S \in \Sigma$ and $p \in {}^{\hookrightarrow}S$, p is stratified.*

In Figure 3.4, any topological sort of the solid edges is a stratified strategy, more efficient than the EXNAÏVE one illustrated in the Figure, since the latter performs four extra transitions. Observe that a stratified strategy does not constrain the order of transitions that *are not on the same path*. For instance, in Figure 3.4, a stratified strategy may apply the transition $S_0 \xrightarrow{\text{JC}} S_1$ before all the SCs.

We now define the important family of EXSTR strategies. Starting from the initial state S_0, an EXSTR strategy picks any state on which it applies any applicable transition, *preserving the stratification of all strategy paths*. Several EXSTR strategies may exist for a workload, differing in their ordering of the transitions. We will simply use EXSTR to refer to any of them. The EXNAÏVE strategy (Algorithm 2) can be turned to an EXSTR one through the following modification: when $applyTrans$ (line 4) is called on a state S_c, it should apply the transitions in a stratified way, i.e., first it attempts a VB and only if no new state is obtained, it applies an SC, and then a JC and, finally, a VF.

Theorem 3.5.3 (Interest of EXSTR)**.** *(i) Any* EXSTR *strategy is exhaustive. (ii) For a given workload Q, and arbitrary* EXSTR *strategy Σ_S and* EXNAÏVE *strategy Σ_N, Σ_S has at most the number of transitions of Σ_N.*

Proof. Exhaustiveness of EXSTR strategies follows from the fact that any state can be reached by a stratified path (Theorem 3.5.2), and that EXSTR only stops when no more states can be discovered. Moreover, Σ_S disables some transitions by restricting the outgoing transitions of a state S depending on S's incoming paths $^{\hookrightarrow}S$, whereas Σ_N allows these transitions. □

Due to Theorem 3.5.3, among the exhaustive strategies, we will only consider *wlog* the stratified ones.

Size of the search space We quantify the size of the search space by the number of states that can be reached through our transitions for a given query workload Q. Due to Theorem 3.5.1, this number is equal to the number of all possible states for Q.

We start the analysis by considering that Q consists of a single query q of n atoms. Consider a possible state $S(Q) = \langle V, R \rangle$. Every view $v \in V$ participates in the rewriting r of q and can, thus, be embedded into it. This means that the graph of v is a subgraph of the graph of q. Since r is a complete rewriting of q, the union of the nodes of the graphs of the views in V is equal to the node set of the graph of q. In other words, the node sets of the view graphs constitute a cover for the node set of q.

However, the set of nodes of a view graph does not uniquely determine a view: we can have more than one different states that have the same node sets for their views. This occurs because they may have different selection and join edges. For instance, consider the query $q(Y) = t(c_1, X, c_2), t(X, c_3, Y)$ and the view sets $V_1 = v_1, v_2$ and $V_2 = v_3, v_2$, where $v_1(X_1) = t(c_1, X_1, c_2)$, $v_2(X_2, Y) = t(X_2, c_3, Y)$ and $v_3(X_1) = t(c_1, X_1, Z)$. In this case, the graphs of the views of V_1 and of V_2 are the same cover of the nodes of the query graph, but the views are not the same (v_1 has one more selection edge than v_3).

To this end, we define a restricted class of states C_r, the view sets of which can be exclusively determined by their node sets. Assume a cover of the node set of the query graph. For each subset of nodes N_s in this cover, we construct a graph G_s which also has N_s as its node set. As for the edge set of G_s, it includes all the selection edges of the query graph that are incident to the nodes of N_s, as well as all the join edges between these nodes. Notice that there are some cases in which the graph that is created is not connected. These graphs correspond to views with Cartesian products and, thus, do not lead to valid rewritings in our setting. However, in the worst case that a query is a clique, every possible subset of its nodes represents a connected graph. Moreover, the cover of the query node set has to be minimal, otherwise our rewritings will not be minimal.

Observe that the states in C_r are those obtained by applying only VBs and from the JCs only those that disconnect the graph of a view. They are the most restricted states (having views with the biggest number of selection and join edges) given a specific cover of the query nodes, as no additional SCs or JCs are applied on them.

Given the above, the problem of enumerating the states in C_r is reduced to the problem of finding the minimal covers of the query node set. Hence, the number of states in C_r is bound by the number of minimal covers.

Let $\mu(n, k)$ be the number of minimal covers with k members of a set of n elements,

given by the following formula:

$$\mu(n,k) = \frac{1}{k!}\sum_{m=k}^{a_k}\binom{2^k-k-1}{m-k}m!\,S(n,m)$$

where $S(n,m)$ is the Stirling number of the second kind (i.e., the number of ways to partition a set of n objects into m groups) and $a_k = \min(n, 2^k-1)$. The overall upper bound of the number of states in C_r, given one query of n atoms is:

$$NS_r(q,n) = \sum_{k=1}^{n}\mu(n,k)$$

For each state $S_r \in C_r$, we can obtain more states through SCs and JCs (those JCs that do not disconnect the graphs). We now need to compute how many such states can we have for a given S_r. To do so, we will first compute the number of nodes of the views in S_r and subsequently the number of selection and join edges. Assume that the query q has n nodes and the view set of S_r comprises k views. In the best case we will have n nodes in total in the views and this happens when all the views were created through JCs (no view node was duplicated). In the worst case, each of the views should cover as many of the query nodes as possible, but without having a view covering a subset of the query nodes of another (due to the need for minimal rewritings). This means that each of the k views should cover $n-k$ query nodes: if a view covered $n-k+1$ then there would exist another view that would cover a subset of the query nodes of this view. For each node we can have at most 3 selection edges. Hence, each of the k views has $3(n-k)$ selection edges. As for the join edges, in the worst case the view will be a clique, having $\frac{(n-k)(n-k-1)}{2}$ join edges. To this end, the k views will have $|e_{sj}| = k\left(3(n-k)+\frac{(n-k)(n-k-1)}{2}\right) = \frac{k(n-k)(n-k+5)}{2}$ edges leading to at most $2^{|e_{sj}|}$ for each S_r state.

We define the class of states C_{re} that includes all the states being obtained from the initial one by applying any possible sequence of VB, SC and JC transitions. For this class we now have:

$$NS_{re}(q,n) = \sum_{k=1}^{n}2^{|e_{sj}|}\mu(n,k)$$

Finally, on each state $S_{re} \in C_{re}$, we can apply a sequence of VFs to obtain new states. In the worst case, any subset of the views in the view set V_{re} of S_{re} consists of isomorphic views. Thus, the number of states resulting from S_{re} by applying VF is bound by the number of partitions of the view set of V_{re}. Assuming the size of V_{re} is k and denoting by B_k the Bell number (the number of partitions of a set of size k), an upper bound for the total number of distinct states belonging to the complete class of states, denoted C_{ref}, is:

$$NS_{ref}(q,n) = \sum_{k=1}^{n}2^{|e_{sj}|}\mu(n,k)B_k$$

We now escalate to the general case when our workload Q consists of n_q queries. Assume each query q_i has n_i atoms, where $1 \le i \le n_q$ and $\sum_{k=1}^{n_q} n_i = n$. Then the total

number of states is:

$$NS_{ref}(Q, n) = \sum_{k=1}^{n_q} NS_{ref}(q_i, n_i)$$

Clearly, the above sum is asymptotically bound by the query with the biggest number of atoms. Thus, the worst case is to have a single query in the workload, because this will lead to the biggest number of atoms per query for a given number of atoms. Thus, in the worst case we have $NS_{ref}(Q, n) = NS_{ref}(q, n)$, where q the single query of the workload.

Time complexity The time complexity of exhaustive search can be derived from the number of states created by each transition and the time complexity of the transition. The cost of a SC, JC and VB is linear in the size of the largest view, which is bound by $3n$, whereas VF requires checking query equivalence, which is in $O(2^n)$ [CM77].

The complexity of exhaustive search is very high and, even if views are selected offline and thus time is not a concern, it brings real issues due to memory limitations. This highlights the need for robust strategies with low memory needs, and efficient heuristics.

3.5.2 Optimizations and heuristics

We now discuss a set of search strategies with interesting properties, as well as a set of pruning heuristics which may be used to trade completeness for search efficiency.

Depth-first search strategies (DFS) A (stratified) strategy Σ is depth-first iff the order of Σ's transitions satisfies the following constraint. Let S be a state reached by a path p of the form VB*. Immediately after S is reached, Σ enumerates all states recursively attainable from S by SC only. This process is then repeated with JC and then with VF. The pseudocode of DFS can be obtained by replacing lines 3-4 of Algorithm 2 with the following ones, where $recApplyTrans$ returns all states that can be reached by a specific transition starting from a given state:

---

foreach *state* $S_{\text{VB}} \in \{recApplyTrans(\text{VB}, S_0)\}$ **do**
  foreach *state* $S_{\text{SC}} \in \{recApplyTrans(\text{SC}, S_{\text{VB}})\}$ **do**
    foreach *state* $S_{\text{JC}} \in \{recApplyTrans(\text{JC}, S_{\text{SC}})\}$ **do**
      foreach *state* $S_{\text{VF}} \in \{recApplyTrans(\text{VF}, S_{\text{JC}})\}$ **do**
        ⋯

For instance, in Figure 3.4, the following strategy Σ_3 is DFS:

$$\Sigma_3 = (S_0 \xrightarrow{\text{SC}(c_1)} S_2), (S_2 \xrightarrow{\text{SC}(c_2)} S_4), (S_4 \xrightarrow{\text{JC}} S_7),$$
$$(S_7 \xrightarrow{\text{VF}} S_8), (S_0 \xrightarrow{\text{SC}(c_2)} S_3), (S_3 \xrightarrow{\text{JC}} S_6)$$

An advantage of DFS strategies is that they fully explore each obtained state more quickly, reducing the number of states stored in CS. This results in a significant reduction of

the maximum memory needs during the search, compared, e.g., with EXNAÏVE, which develops a huge number of candidates before fully exploring them.

Aggressive view fusion (AVF) This technique can be included in any strategy and is based on the fact that VF can only decrease the overall cost of a state (Section 3.3.4). Once a new state S is obtained through some SC, JC or VB, we recursively apply on S all possible VFs (until no more views can be fused). It can be shown that such repeated VFs converge to a single state S^{VF}. We then discard all intermediate states leading from S to S^{VF} and add only S^{VF} to CS. Thus, AVF preserves the optimality of the search, all the while eliminating many intermediary states whose estimated cost is guaranteed to be higher than that of S^{VF}. For example, assume we reach a state S containing three identical views. We apply a VF on S fusing two of the three views and obtain the state S'. We then apply a VF on S' fusing the two remaining identical views and obtain S^{VF}. AVF discards S' and keeps only S^{VF} to continue the search.

Greedy stratified (GSTR) This strategy starts by applying all possible VB transition sequences on S_0. It then discards *all the obtained states but* S_b, and repeatedly applies on it all possible SC. Keeping only S_b, it proceeds in the same way by applying JC and then VF. The interest of GSTR lies in the possibility to combine it with the AVF technique, leading to the GSTR-AVF strategy. GSTR-AVF has low memory needs due to the many states dropped by GSTR and AVF, and moves fast towards lower-cost states due to AVF. Although neither GSTR nor GSTR-AVF can guarantee optimality, they perform well in practice, as our experiments show.

Stop conditions We use some *stop conditions* to limit the search by considering that some states are not promising and should not be explored. Clearly, stop conditions lead to non-exhaustive search. We have considered the following stop conditions for a state S.

- $stop_{tt}(S)$: true if a view in S is the full triple table t.
- $stop_{var}(S)$: true if a view in S has only variables. The idea is that we reject S since we consider its space occupancy to be too high. In general, this condition may be satisfied even by the initial state S_0. In this case, $stop_{var}$ would prevent *any* search. However, such queries are of very limited interest. Therefore, $stop_{var}$ can be used in many settings to restrict the search while leaving many meaningful options.
- $stop_{time}(S)$: true if the search has lasted more than a given amount of time. Observe that our approach is guaranteed to have *some* recommended S_b state at any time.

Pull&push constants technique (PPC) This technique makes *educated guesses* on which selection edges to cut and which to preserve. It orders all constants from the workload, according to their number of occurrences. The more frequent the constant, the more likely it is to appear in the selected view state, because it represents a selective condition shared by many views. Thus, prior to any search, we cut *all* selection edges corresponding to constants appearing one or very few times ("pull constants" part), since these constants will most probably not appear in a low-cost state. If this pre-processing removes l selection edges, this diminishes the search space by a significant factor of 2^l, given that the subsequent search (regardless of its strategy) will be applied on an initial CS of just one state (obtained from S_0 by the l successive SCs). *After* the search has finished, however, we may be able to "push" back some of the constants cut in the "pull" stage. This is the

case if, for a recommended view v, *all* rewritings using v apply the same selection on v, corresponding to a constant eagerly removed by the "pull".

PPC may compromise optimality, given that the comparisons performed during the search ignore the fact that some selections may be brought back by the post-processing.

For example, consider the following two simple queries:

$q_3(X_1, X_2) :- t(X_1, hasTitle, X_2), t(X_1, type, paint)$
$q_4(X_3) :- t(X_3, hasTitle, starryNight)$

If we apply PPC by *pull*ing some of the constants that appear only once in the workload, namely $paint$ and $night$, our initial state becomes:

$v_3(X_1, X_2, X_4) :- t(X_1, hasTitle, X_2), t(X_1, type, X_4)$
$v_4(X_3, X_5) :- t(X_1, hasTitle, X_5)$

Assume the best state of the search is obtained after applying a JC on v_3 (leading to two new views v_5 and v_6) and a VF between v_5 and v_4. The views of the state are:

$v_6(X_1, X_4)$:-$t(X_1, type, X_4)$, $v_7(X_6, X_7)$:-$t(X_6, hasTitle, X_7)$

The rewritings are (projections are omitted for readability):

$q_3 = \sigma_{X_4=paint}(v_7 \bowtie_{X_6=X_1} v_6)$, $q_4 = \sigma_{X_7=night}(v_7)$

In this case, removing the constant $night$ was a good choice, as it enabled a VF. However, removing $paint$ did not help and this constant can be *push*ed back, creating the view $v_8(X_1)$:-$t(X_1, type, paint)$ which will be used instead of v_6 in the rewriting of q_3. The resulting state with $paint$ pushed back has a lower cost, since v_8 occupies less space than v_6, and the rewriting of q_3 is also more efficient (a simple scan on a smaller table).

3.6 The RDFViewS System

To assess the interest of our view selection approach in practice, we have built the RDFViewS system [GKLM10a], standing for *RDF View S*election. RDFViewS focuses on automatically choosing the materialized views that are most appropriate for a given dataset and query workload. All our techniques for exploring possible view configurations and taking into account the implicit data brought by an RDFS, presented in Sections 3.5 and 3.4, respectively, of this Chapter, have been implemented and incorporated in RDFViewS.

The system takes as input a workload of conjunctive SPARQL queries and possibly an RDFS, and outputs a set of views to be materialized, along with the rewritings of the workload queries based on this view set. It uses a relational database back-end to store the original RDF data, as well as the materialized views. RDFViewS can be seen, in effect, as a means to tune an RDF database based on the knowledge of the query workload, to be used in conjunction with off-the-shelf RDBMSs. The basic choices we made with respect to the platform and the data storage are discussed in Section 3.6.1, whereas Section 3.6.2 presents the architecture of the system.

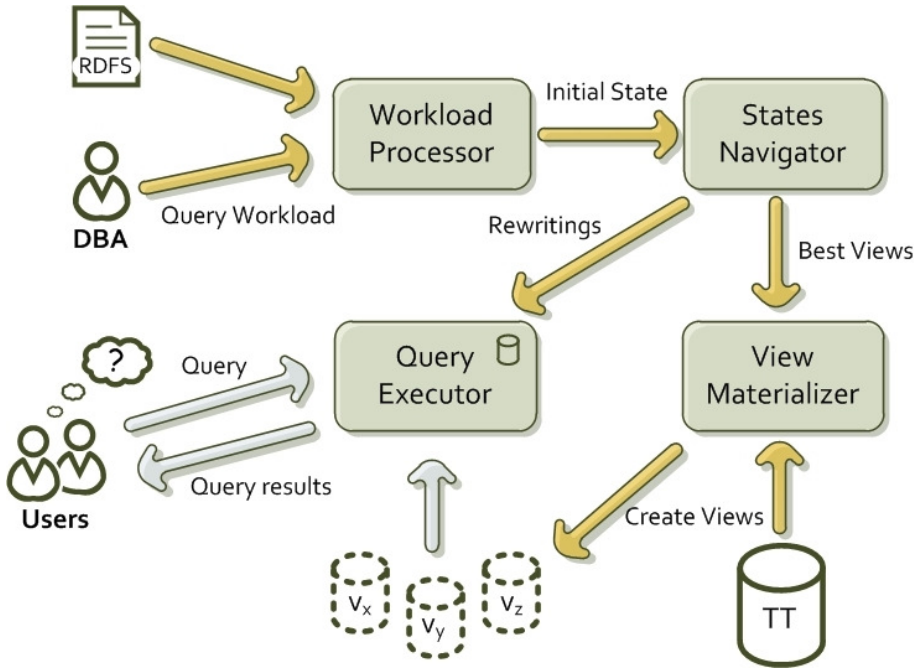


Figure 3.5: RDFViewS architecture.

3.6.1 Platform and Data Storage Details

RDFViewS has been fully implemented as a Java 6 application. To store our data and views, and facilitate the evaluation of the rewritings, we needed a database back-end of reasonable scalability. To that end, we chose PostgreSQL (version 8.4.3), both for its reputation as a (free) efficient platform, and because it has been used in several related works [AMMH07, NW08, NW09, SGK+08, WKB08]. Integrating our view selection approach with another platform is easy, as soon as that platform supports the evaluation of our select-project-join rewritings, and provided that the cost function is appropriately customized to account for the respective evaluation engine.

As in many previous works, for efficiency, we stored the data in a dictionary-encoded triple table, using a distinct integer for each distinct URI or literal appearing in an s, p or o value. The encoding dictionary was stored as a separate table indexed both by the integer dictionary code and by the encoded constant. The triple table was clustered by the columns p and then s, to enhance the efficiency of (frequent) queries where the p values are specified in most or all atoms. Moreover, we indexed the encoded triple table on s, p, o, and all two- and three-column combinations.

3.6.2 System Architecture

The architecture of RDFViewS is depicted in Figure 3.5. The RDF data is initially stored into an RDBMS as a single triple table (*TT*). The conjunctive SPARQL query workload is provided as input to the **Workload Processor** module, through a GUI. The

query workload is then used to create the initial state of the search (see Section 3.5.1).

Subsequently, the initial state is loaded to the **States Navigator** module, which incorporates all the search strategies and heuristics, presented in Section 3.5. Through the GUI, the user can select the search strategy that better meets her needs. Moreover, she can tune the weights of the cost function (Section 3.3.4), based on the importance of each cost component (query execution time, view storage space or view maintenance cost) for the specific application. Then, the search for the best state (view configuration) is performed, according to the desired search strategy and cost estimation.

Once this search has finished, the **View Materializer** module materializes the chosen set of views, after translating them to SQL. Then, the rewritings contained in the best state are pushed to the **Query Executor**, which stores them for future use. Whenever a user issues a query from the input query workload, the Query Executor uses the stored rewritings to efficiently answer the query by using the already materialized views.

In case an RDFS is also given as input to the system, it is taken into account by employing one of the approaches discussed in Section 3.4: in database saturation the implicit triples are added to the database, in pre-reasoning the queries are reformulated prior to the search, whereas in post-reformulation the views of the chosen state are reformulated.

3.7 Experimental Evaluation

To experimentally evaluate our approach, we used our RDFViewS system, which was presented in the previous Section. Below we give some more details about our experimental setting.

Data and queries As in previous works [AMMH07, NW08, WKB08], we used the Barton RDF dataset and RDFS [www]. The initial dataset consists of about 50 million triples. After some cleaning (removing formatting errors, eliminating duplicates etc.), we kept about 35 million distinct triples.

The Barton query workload [www] contains few queries with no commonality among them. To better test our approach, we built two query generators, producing queries of controllable size, shape, and commonality. The first one simply outputs the desired queries, and has maximum flexibility. The second takes as input not only the workload characteristics, but also a dataset (RDF + RDFS) and generates queries having non-empty results on the given dataset. We used it to obtain interesting workloads on the Barton dataset.

Weights of cost components For VSO and REC (Section 3.3.4), we used c_s=1 and c_r=1. For each workload, we set the value of c_m taking into account the database size and the average number of atoms in each query, so that for the initial state S_0, $c_m \cdot VMC$ is within at most two orders of magnitude from the other two cost components, namely $c_s \cdot VSO$ and $c_r \cdot REC$. In most cases, this lead to c_m=0.5. Finally, we set f=2 in VMC, since this value gave the most appropriate range to VMC throughout the search.

Hardware and memory The PostgreSQL server ran on a separate 2.13 GHz Intel Xeon machine with 8GB RAM. We ran the search algorithms on two classes of hardware: a *desktop* 8-core Intel Xeon 2.13 GHz machine with 16 GB RAM (the JVM was given 4

GB), and several *cluster machines*, each of which was a 4-core Intel Xeon 2.33 GHz with 4 GB RAM (the JVM was given 3 GB). Each experiment ran on one machine. While there are opportunities for parallelization (see Section 3.9), we did not exploit them in this work. All machines were running Mandriva Linux 2.6.31.

In the sequel, we briefly present the search strategies used in the relational approach [TLS01] in Section 3.7.1, and compare with them in Section 3.7.2. In Section 3.7.3 we highlight the impact of our heuristics in the search, and give results on the achieved cost reductions for large workloads in Section 3.7.4. Section 3.7.5 assesses the performance of our reasoning approaches, then Section 3.7.6 reports on the impact of our view selection approach on query evaluation, and, finally, Section 3.7.7 provides more details on the used cost components.

3.7.1 Competitor search strategies

We have implemented the three strategies, *Pruning*, *Greedy* and *Heuristic*, introduced in the relational view selection work which inspired our states and transitions [TLS01]. All these strategies follow a divide-and-conquer approach. They start by breaking down the initial state into a set of one-query states, and apply all possible edge removals, then all possible view breaks on each such state. Then, they seek to put back together states corresponding to the complete workload by adding up and, when appropriate, fusing, one state for each workload query. Since any combination of partial states leads to a valid state in [TLS01], the number of states thus created explodes. To avoid it, *Pruning* discards partial states outgrowing the given space or cost budget, whereas *Greedy* develops very few states: it only keeps the best combined state, say, for the workload queries $\{q_1, q_2\}$, even though this may prevent finding the best combined state for $\{q_1, q_2, q_3\}$. Finally, *Heuristic* resembles *Pruning*, except that after having built all one-query states, it only keeps: the minimal-cost state for each query, and any states which offer some view fusing opportunity. Since our algorithms do not use a cost or space budget, we did not give one to the [TLS01] strategies either. This does not prevent their pruning which is mostly based on comparing two states and discarding the less interesting one.

Search strategy acronyms In the sequel, for convenience, we will refer to the [TLS01] strategies simply as *Pruning*, *Greedy* and *Heuristic*. Among the strategies we propose (see Section 3.5.2), DFS is the (stratified) depth-first search, while GSTR is the greedy strategy. The suffixes -AVF and -PPC after a strategy name denote aggressive view fusion and pull&push constants, respectively, applied in conjunction with that strategy. The suffix -STV denotes that the $stop_{var}$ stop condition is used, while -PPC-k, where k is an integer, denotes the pull&push constants optimization that removes all selection predicates on constants that appear less than k times in the workload.

Relative cost reduction To assess search effectiveness, we define the *relative cost reduction (rcr)* of a given strategy Σ and workload Q, as the ratio $(c^\epsilon(S_0) - c^\epsilon(S_b))/c^\epsilon(S_0)$ at a given moment, that is, the fraction of the cost of the initial state S_0, avoided by the current best state found by Σ by that moment during the search.

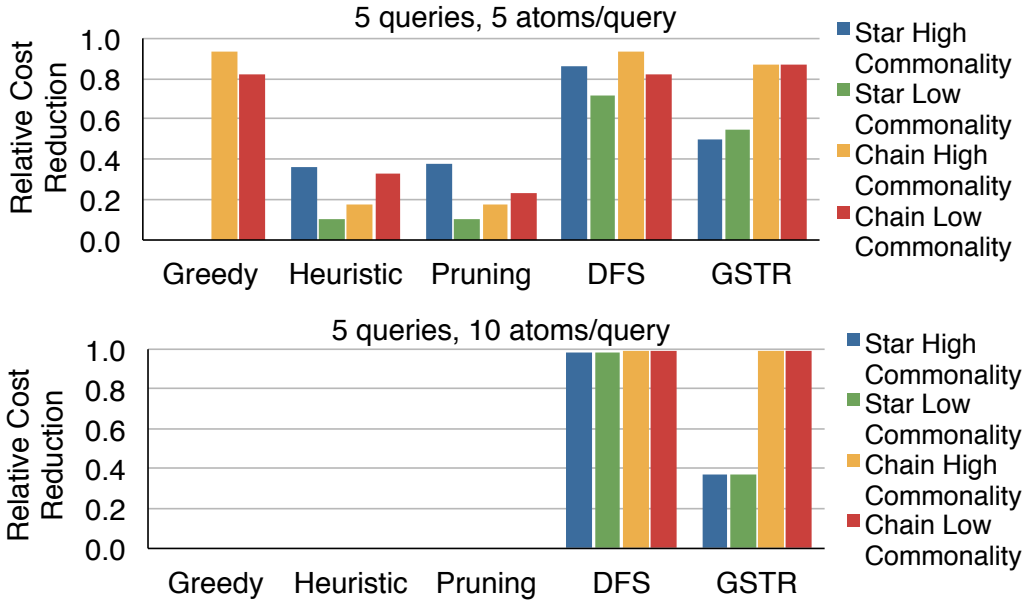


Figure 3.6: Strategy comparison on small workloads.

3.7.2 Comparison with existing strategies

We compare our strategies with those of [TLS01] for two small workloads of 5 queries each. While the queries they tested involve on average 4 relations, one needs more RDF atoms than relations to express the same logical query, since data that would fit in a wide relational tuple is split over many RDF triples. Thus, queries in the first and second workload have 5 and 10 atoms each, respectively.

Figure 3.6 shows the rcr of the three strategies of [TLS01] and our strategies DFS-AVF-STV and GSTR-AVF-STV. The reasons for using the specific heuristics on our strategies are explained in Section 3.7.3. The Figure considers workloads of star and chain queries, which are typical in RDF. In particular, star queries translate to query graphs (Definition 3.3.1) that are cliques (each atom is connected to all others), allowing for many VBs and JCs and, therefore, have a search space of increased size, whereas chain queries can be considered an average case regarding the difficulty of the search. The workloads were generated both with high and low commonality across queries and we used the $stop_{time}$ stop condition, set to 30 minutes. While this may seem long, recall that the complexity of search is high (Section 3.5.1); we consider this duration acceptable as view selection is an *off-line* process. The overhead is worth it especially for large workloads, and/or queries asked repeatedly.

As can be seen in Figure 3.6, for the smaller workload, all strategies ran well, with DFS-AVF-STV and GSTR-AVF-STV being the best. The runs did not finish, i.e., the strategies might have found better solutions by searching longer. *Greedy* managed to reduce the cost significantly for chains but failed to find any state better than the initial one for stars queries. For the larger workload, the [TLS01] strategies failed to produce any solution, as they outgrow the available memory building *partial* states (for $1, 2, 3$ queries

etc.) before building *any* state covering all 5 queries. In contrast, DFS-AVF-STV and GSTR-AVF-STV keep running and achieve interesting cost reductions. The same trend was observed on workloads with cycle- and random graph-shaped queries (we generated both sparse and dense graphs), at high and low commonality.

Thus, from now on, and in particular for large workloads, we focus only on our strategies, since those of [TLS01] systematically outgrow the memory before reaching a full candidate view set.

3.7.3 Impact of heuristics and optimizations

We now study the impact of the AVF, STV and PPC techniques on the search space explored by our algorithms. Tiny workloads of 2 queries of 4 atoms each, suffice to illustrate this. We used the DFS and GSTR strategies with several combinations of heuristics and compared them with the 3 algorithms of [TLS01]. Figure 3.7 shows numbers collected with the DFS strategies on star-shaped, chain-shaped and mixed query workloads. Figures 3.8 and 3.9 show results obtained with the same workloads running the GSTR strategy and the algorithms of [TLS01] respectively. The created states are those reached by the search. States previously attained through a different path are recognized by searching for their presence in the ES or CS sets (Section 3.5), considered *duplicates* and ignored. The STV heuristic leads to *discarding* some states. Finally, *explored* states are those from which all outgoing transitions respecting the given strategy have been explored.

A first remark based on Figure 3.7 is that the number of duplicate states may be quite important. Duplicates occur because even when using a stratified strategy, a state may be reached by more than one path. For instance, assume for some given views v_1, v_2 that an SC modifies v_1 into v_1' (denoted $v_1 \xrightarrow{\text{SC}(c_1)} v_1'$) and similarly $v_2 \xrightarrow{\text{SC}(c_2)} v_2'$. From the state (v_1, v_2), our algorithms reach the state (v_1', v_2') twice: once through (v_1, v_2') and a second time through (v_1', v_2). Our algorithm identifies such states as soon as they are created, in order not to repeat their exploration.

AVF alone has marginal effect on the number of states explored, but STV prunes the space very efficiently regardless of the workload type. As for PPC-1, its impact is not always beneficial w.r.t. the search space size. Although it dramatically cuts down the total search space size of a star-shaped query workload, using PPC on chain-shaped queries does not significantly reduce the number of duplicates reached. Chains generally contain less constants than stars, thus attempting to reduce the space size simply by removing constants has a mild effect. In this case, PPC-1 reduced the total search space size only by 21%.

In Figure 3.8, the GSTR strategy is used in conjunction with AVF, since we observed through our experiments that AVF when used with GSTR significantly improved the cost of the best returned state (not waiting until the last stage of GSTR to perform VFS, increases the chances to find a better state). As can be seen from the Figure, GSTR overall generates much less states than DFS. The STV heuristic did not have an impact in these cases, because it seems that the states discarded from this heuristic were not among those states that GSTR was going to continue the search upon. As for PPC-1, it reduced the number of explored states for stars, but had no significant impact on the chain-query

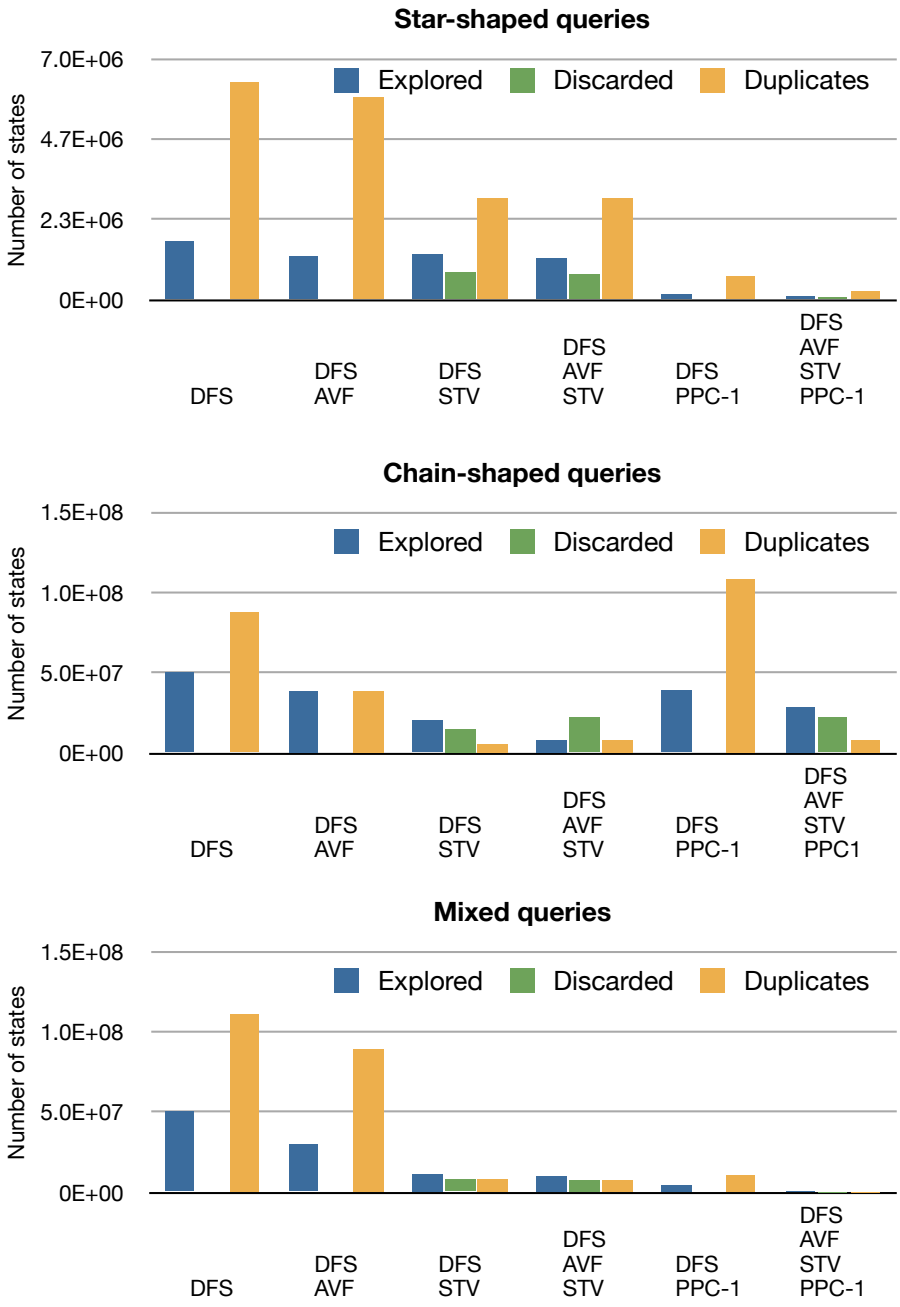


Figure 3.7: Impact of heuristics on the DFS strategy for star, chain and mixed query workloads.

workload.

Let us now examine the search space size for the three strategies of [TLS01] (Figure 3.9). Due to the differences in the nature of the searches, we cannot directly compare the number of states created by those strategies with the ones created by ours. In particular, for the strategies of [TLS01], we measure the one-query states and the final states. As

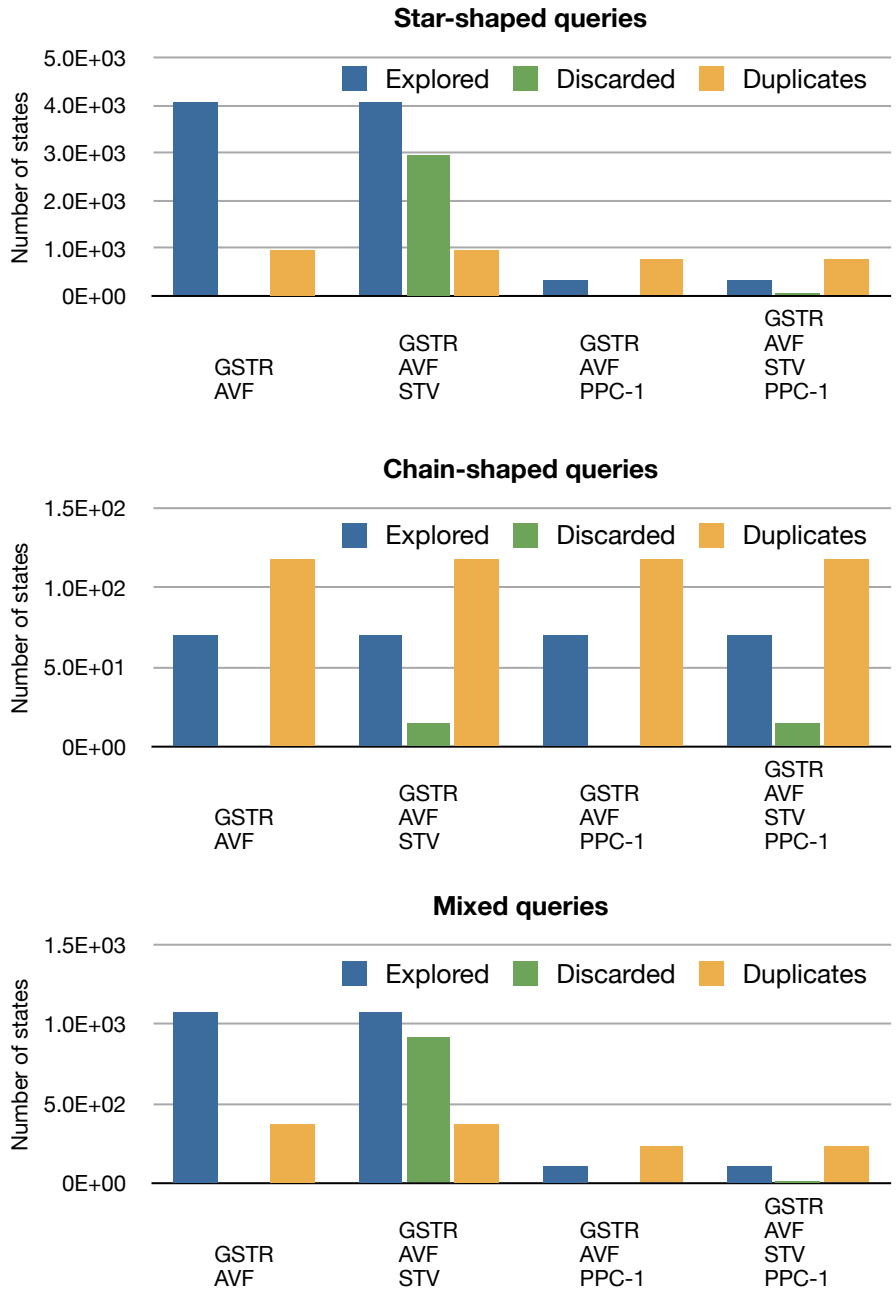


Figure 3.8: Impact of heuristics on the GSTR strategy for star, chain and mixed query workloads.

explained in Section 3.7.1, all three strategies first create one-query states (states for each of the queries of the workload) and then combine them to reach final states (which contain rewritings for all the queries). As expected, all three strategies generate the same number of one-query states, as in this stage no pruning has been performed yet. The *Greedy* strategy has only one final state, as it only picks each time the best state (resulting from a

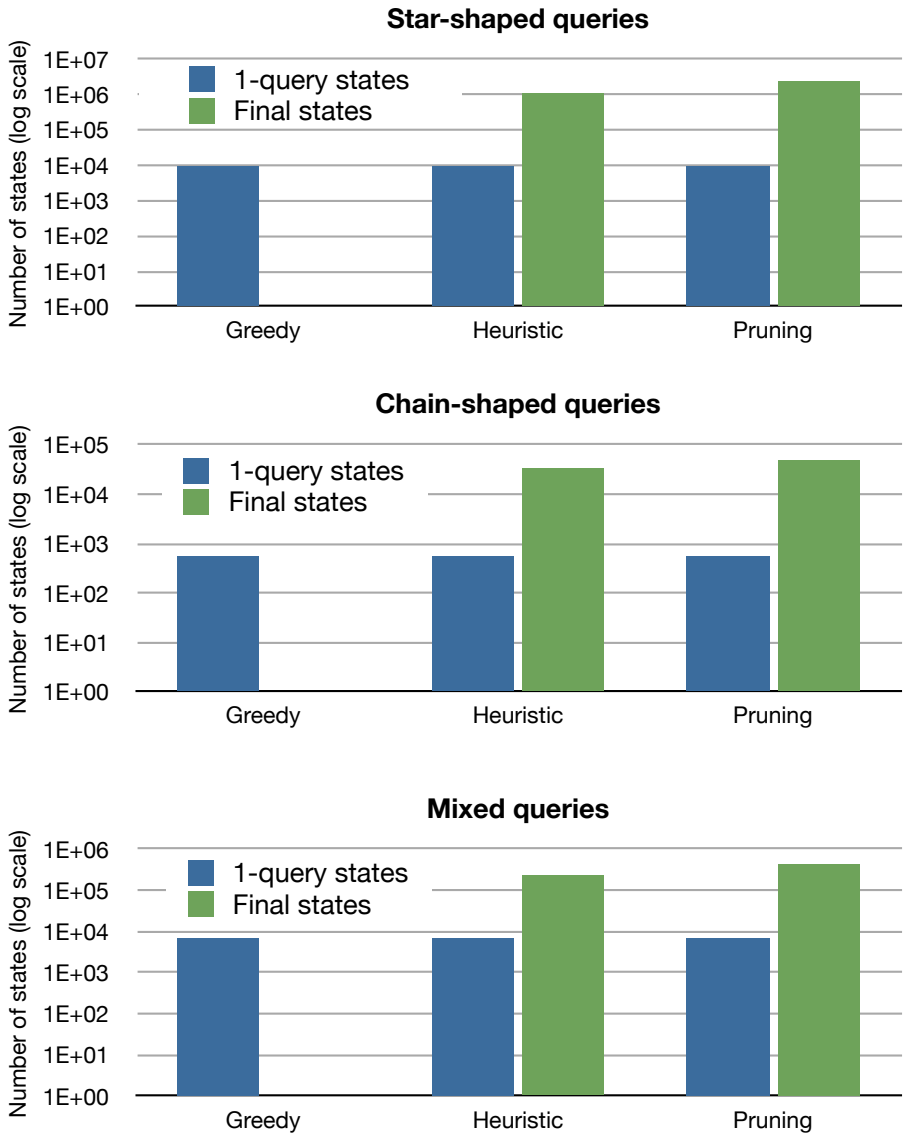


Figure 3.9: Impact of heuristics on the *Greedy*, *Heuristic* and *Pruning* strategies for star, chain and mixed query workloads.

combination of one-query states) to continue the search. Between *Pruning* and *Heuristic*, we observe that the additional heuristic criterion that the latter uses, indeed contributes in generating less final states.

Optimality of search strategies We also studied the evolution of the best cost with the search time both for DFS and GSTR with various combinations of heuristics. Figures 3.10 and 3.11 show the results of this experiment. We recall that DFS and DFS-AVF are optimal (i.e., they find the globally optimal state).

The experiment illustrates, first, that GSTR explores much less states and therefore is much faster (3 orders of magnitude in this example) than DFS. On this example which was

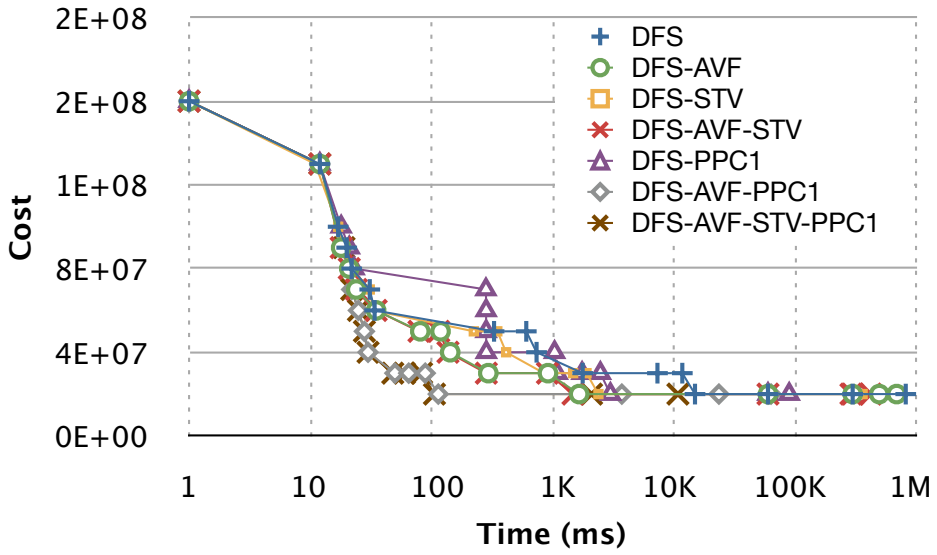


Figure 3.10: Cost reduction over time with an exhaustive DFS strategy and various combinations of heuristics.

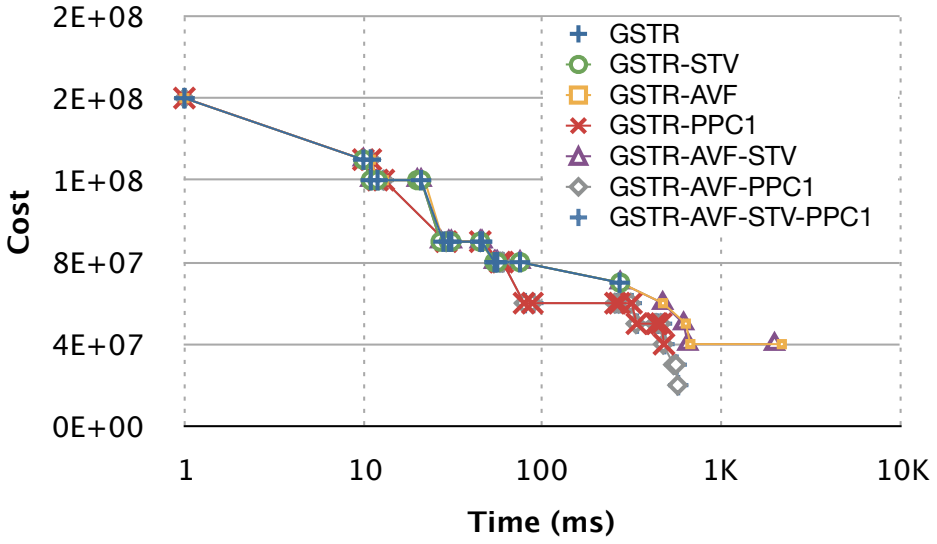


Figure 3.11: Cost reduction over time with GSTR strategy and various combinations of heuristics.

purposely small, all DFS variants converged to the same globally optimal state, although this cannot be guaranteed in general. Observe, however, that the final state reached by GSTR versions is not the optimal one, reached by plain DFS. In particular, on this example the ratio between the cost of the optimal state (found by DFS) and that of the best state obtained by GSTR ranges from 0.28 to 0.99, depending on the heuristics used. Again, this ratio holds for this specific example and depends on the given workload.

Among the DFS variants, we notice that applying AVF has a consistently good impact,

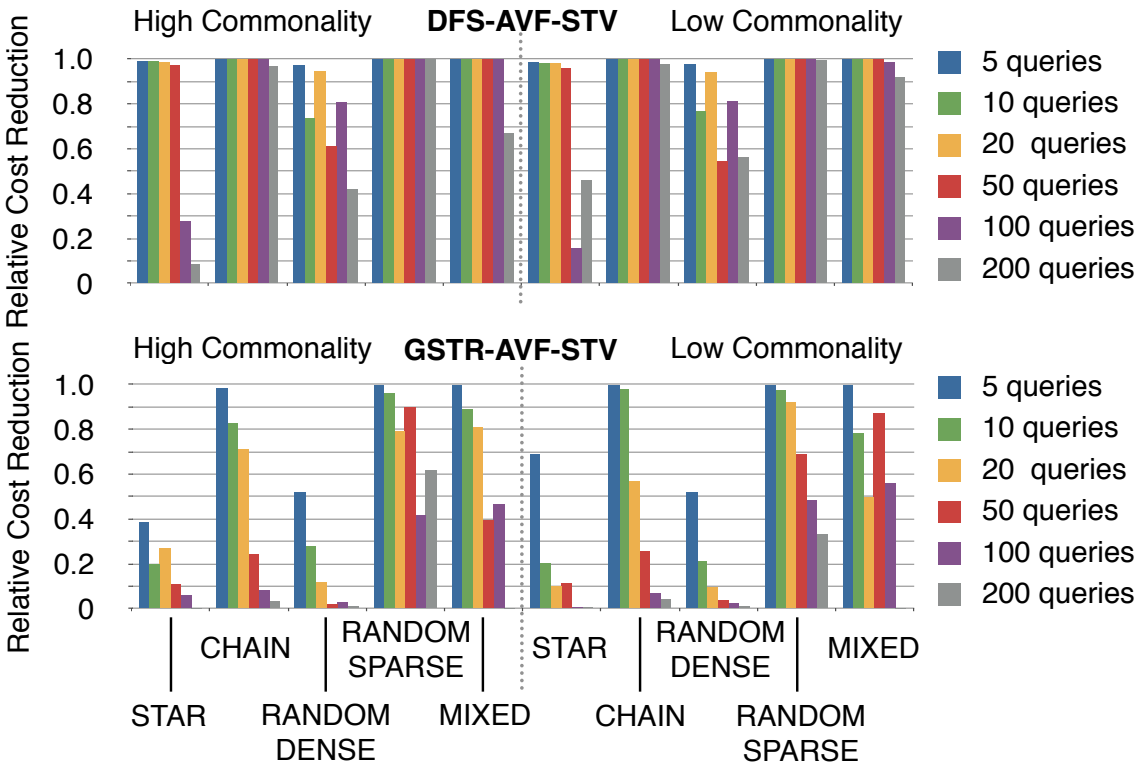


Figure 3.12: Relative cost reduction for large workloads.

i.e., the cost of the best state found decreases more quickly when AVF is enabled. Hence, by using AVF, we can stop the search earlier and get a better result than the plain DFS would give at the same point. The combination of all heuristics yielded the best results in this example, although in general, PPC and STV may compromise the quality of the best state in exchange for shortening the search.

Among the GSTR variants, an interesting remark is that the PPC heuristic leads to attaining a better final state than if PPC is not used. This can be explained as follows. Applying PPC at the very beginning of the search leads to a state from which the heuristic exploration of GSTR turned out to find a better final state. However, this cannot be guaranteed in general.

In the sequel, given the above results, we will systematically use the combination AVF-STV both for DFS and GSTR, since these heuristics make the search significantly faster in most cases, without returning a state with a significantly bigger cost.

3.7.4 Cost reduction on large workloads

We study the scalability of our DFS and GSTR algorithms for large query workloads. To this purpose, we generated workloads of 5, 10, 20, 50, 100 and 200 queries; each query has 10 atoms, i.e., the views of the initial states contain 10 atoms on average. We consider

Workload Q	$\|Q\|$	$\#a(Q)$	$\#c(Q)$	$\|Q^r\|$	$\#a(Q^r)$	$\#c(Q^r)$
Q_1	5	33	35	20	143	157
Q_2	10	76	77	231	1436	1651

Table 3.3: Workloads used for reformulation experiments.

workloads consisting of: star queries only; chain queries only; random-graph shaped queries (with two variants, dense graph and sparse graph); mixed, combining queries of all previous shapes. For each kind of workload, we generate three low- and three high-commonality variants. On each of these 30 workloads, we ran DFS-AVF-STV and GSTR-AVF-STV. We used the $stop_{time}$ stop condition set to 3 hours. These experiments ran in the cluster.

Figure 3.12 plots for each of the 10 workload types, the rcr averaged over the 3 workloads of that type, at the end of the search. A first remark is that DFS's relative cost reduction is very impressive overall, and in many cases around 0.99. Second, note that the rcr of GSTR-AVF-STV is generally smaller than that of DFS-AVF-STV, because GSTR explores significantly fewer states than DFS and might miss interesting opportunities. Third, we can distinguish "easier" workloads, such as chains and random-sparse graphs, resulting in query graphs with fewer edges and, thus, fewer transitions. For such workloads, the rcr is higher since the search space is smaller (and bigger part of it was explored). Stars and random-dense graphs are difficult cases, as they lead to many edges, thus smaller rcrs. Finally, the rcrs obtained for high-commonality workloads are generally higher than for low-commonality, e.g., for random-dense and mixed workloads. This confirms the intuition that more factorization opportunities lead to higher gains.

We conclude that DFS-AVF-STV scales well up to 200 queries, depending on the workload structural complexity, and can achieve very significant reductions in the state cost.

3.7.5 View selection and implicit triples

We now study the impact of implicit triples on view selection performance. Starting from a non-saturated database D and workload Q, three scenarios are possible: (i) saturated database D^s, search on Q, using the statistics of D^s; (ii) original database D, search on the pre-reformulated workload Q^r, using the statistics of D; (iii) original database D, search on Q, using the statistics of the saturated database D^s (recall from Section 3.4.3 that we gather them *without* actually saturating the database). Of course, we consider the same RDF entailment rules for the three scenarios, i.e., those brought by an RDFS. Saturation and post-reformulation coincide for any search algorithm, since they lead to the same input statistics and workload. Hence, we only study the search for pre- and post-reformulation.

This experiment uses the Barton dataset as well. The schema consists of 39 classes, 61 properties, 24 rdfs:subClassOf statements, 2 rdfs:subPropertyOf statements, 39 rdfs:domain statements and 41 rdfs:range statements. We generated two satisfiable workloads Q_1 and Q_2, whose properties and those of their reformulated versions Q_1^r and Q_2^r

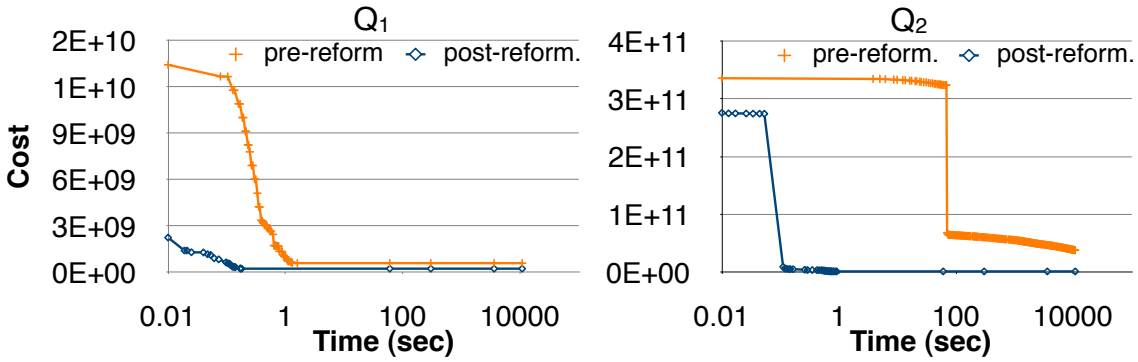


Figure 3.13: Search for view sets using reformulation.

are given in Table 3.3. $|Q|$ denotes the number of queries in Q, $\#a(Q)$ the number of atoms, and $\#c(Q)$ the number of constants. Q_1 is a subset of Q_2.

Figure 3.13 shows the evolution of the best cost found by DFS-AVF-STV for both workloads (post-reformulation) and their reformulated variants (pre-reformulation). The search was cut after 3 hours. We see that the initial state for reformulated workloads has higher cost than the original workloads. Further, the best state cost decreases rapidly with post-reformulation, because the workload is much smaller and the search space is traversed faster. In contrast, the important workload sizes slow down the cost decrease for pre-reformulation. At the end of the search, pre-reformulation's best cost found is higher than that of post-reformulation, by a factor of 2.7 for Q_1, and 22 for Q_2. This confirms our expectation that the advantages of post-reformulation are most visible for larger workloads (with larger Q^r). Moreover, the best cost is reached faster in post-reformulation.

In general, the number of implicit triples increases with the size of the database D and of the Schema $\mathcal{S}$. Let $|D|$ be the number of triples in the database and $|\mathcal{S}|$ the number of statements in the RDFS. Given the statements allowed in an RDFS (Table 3.1), in the worst case each triple can yield two triples by a first application of the rules, resulting in $2 \cdot |D|$ implicit triples. This is the case when each of the triples uses a property whose domain and range is the same. In fact, if triple $t = (u_1, p, u_2)$ appears in D, and $(p, \text{rdfs:domain}, c)$ and $(p, \text{rdfs:range}, c)$ appear in $\mathcal{S}$, t leads to two implicit triples: $t' = (u_1, \text{rdf:type}, c)$ and $t'' = (u_2, \text{rdf:type}, c)$. In turn, each of the t' and t'' can yield new triples through subclass statements. In the worst case, there are $2 \cdot |\mathcal{S}|$ distinct classes in $\mathcal{S}$ (two classes participate in each statement, and the bound of $2 \cdot |\mathcal{S}|$ classes is reached when each class appears only in a single statement in $\mathcal{S}$). Thus, it follows that the size of the saturation is in $O(|D| \cdot |S|)$.

Similarly, $|Q^r|$ may be exponentially larger than $|Q|$ (see Theorem 3.4.1). *In a reformulation-based setting*, view selection based on post-reformulation is clearly better than based on pre-reformulation, since the initial state is better and search is faster, especially for large workloads. *Among saturation and post-reformulation*, the best choice strongly depends on the context (distribution, rights to update the database, frequency and types of updates, etc.) as explained in Section 3.4.2. The views recommended in a

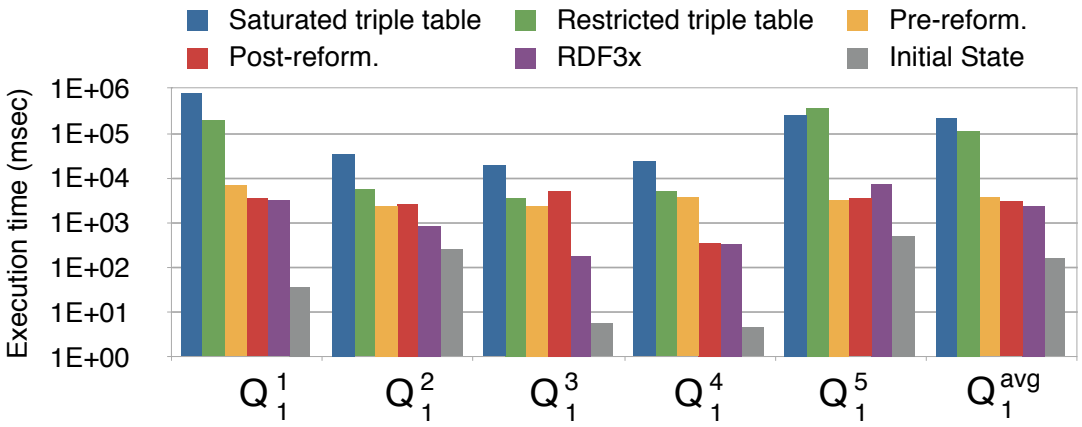


Figure 3.14: Execution times for queries with RDFS.

saturation and a post-reformulation context are the same.

Cost of statistics collection In order to assess the additional cost brought by the statistics collection in each of the three reasoning approaches, we measured the corresponding time needed to gather the statistics for a workload of 10 queries, having a total of 24 atoms on the Barton database. Overall, we found these costs to be acceptable:

- gathering the (workload-relevant) statistics on the saturated database took 59 seconds, while saturating the Barton dataset took 37 minutes;
- gathering the statistics for pre-reasoning took 147 seconds;
- post-reasoning requires estimating the size of a set of atoms: those appearing in the workload, as well as any relaxation thereof (the size of these atoms is counted in all cases, as explained in Section 3.3.4). Recall though that in the post-reformulation approach, the size of this set of atoms must be counted *as if* the database was saturated, without actually saturating it, as explained in Section 3.4.3. Observe that the most relaxed atom t_0, defined by $(?X, ?Y, ?Z)$ (the size of the saturated database) needs to be counted for *any* workload, because it is a relaxation of any possible query atom. In our workload, computing the size of this "most general" term took 849 seconds, which is quite high, but observe that this only needs to be gathered *once for the whole database*, and can be subsequently cached and re-used for various workloads. Computing the size of *all other relaxed atoms* in our example took 391 seconds, which is comparable with the 147 seconds of pre-reformulation. Thus, the overhead of the statistics collection in the case of reformulation is not significant, given the benefit that this approach brings compared to pre-reformulation with respect to the time needed for the actual search.

3.7.6 View-based query evaluation

We now study the benefits that our recommended views actually bring to query evaluation (recall though that our view selection does not optimize for query evaluation *only*, but for a combination including storage and maintenance costs). For the

	DFS	GSTR	TT	DFS^r	GSTR^r	TT^s
Q_3^1	596	600	732	154	623	19282
Q_3^2	559	586	271	76	587	3317
Q_3^3	1798	3277	271	76	3354	95755
Q_3^4	1014	1037	23479	1537	1068	62688
Q_3^5	75	77	1772	146	71	11490

Table 3.4: Execution times (msec) for high commonality workload of 5 queries.

	DFS	GSTR	TT	DFS^r	GSTR^r	TT^s
Q_4^1	90	39	1635	75	121	460
Q_4^2	48	43	4525	31	64	17930
Q_4^3	3014	9210	66	3101	17034	54989
Q_4^4	11	13	367	39	59	238
Q_4^5	43	257	1451	70	158	1192

Table 3.5: Execution times (msec) for low commonality workload of 5 queries.

workload Q_1 described in Section 3.7.5, we materialized the views recommended by pre- and post-reformulation, and ran the 5 queries Q_1^1 to Q_1^5 of Q_1 using (i) the views, (ii) the (dictionary-encoded, heavily indexed) *saturated* triple table in PostgreSQL, (iii) a restricted version of (ii) only with the triples needed for answering Q_1, (iv) RDF-3X [NW10b] (loading the saturated database in it), and (v) the materialization of the query workload (initial state). RDF-3X times were put as a reference; by using PostgreSQL (even with views) we did not expect to get better times than those of a state-of-the-art RDF-specific platform.

The views were materialized in 81 seconds for post-reformulation (the total view size was 433 MB or 15% of the database size), and 103 seconds for pre-reformulation (601 MB or 21% of the database size). Figure 3.14 shows that using our views, queries are evaluated more than an order of magnitude faster than on the triple table, even when using the restricted triple table (iii). Both pre- and post-reformulation performed in the range of RDF-3X. This is a promising result, since our approach can be used on top of RDF-3X and achieve an even bigger gain. Finally, as expected, materializing the queries gives the best results (simply scanning the views is sufficient).

Tables 3.4 and 3.5 show the detailed query evaluation times for 2 additional workloads, Q_3 and Q_4, of high and low commonality respectively, consisting of 5 queries each. The first three columns do not take into account any RDF Schema, while in the last three the Barton RDF Schema was used. Columns DFS and GSTR report the times obtained using the views that were recommended by the respective strategies (note that, as usual, the AVF and STV heuristics were enabled). Column TT refers to the evaluation of the workloads directly onto the triple table. Likewise, columns DFS^r and GSTR^r use the views recommended by the two corresponding strategies through post-reformulation, while for TT^s the saturated version of the triple table was used.

When the view sets recommended by our search strategies are used in answering the

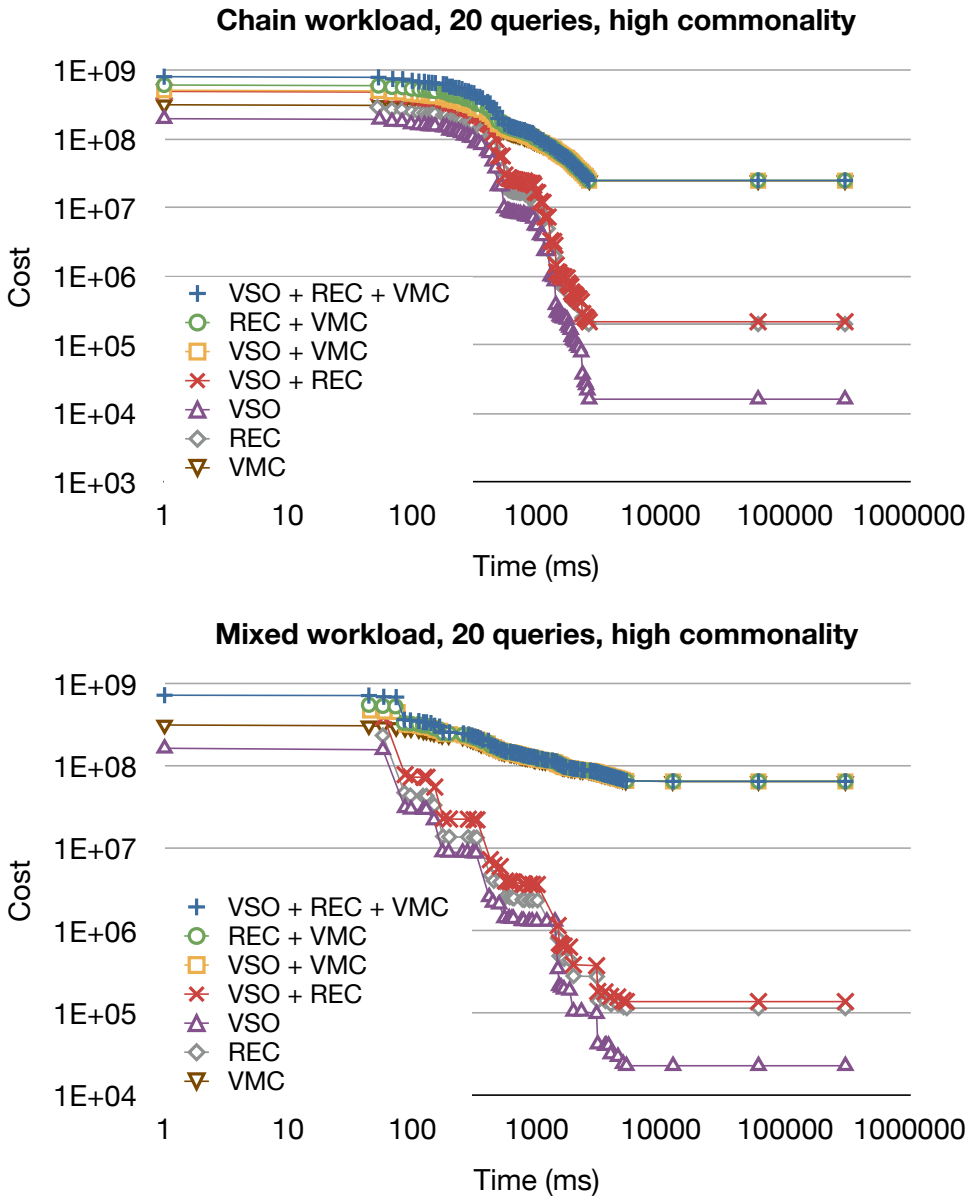


Figure 3.15: Cost reduction with varying cost components.

queries, the evaluation times in most cases are better than when using the triple table. The gain in time though is much more significant when the RDFS is present. These examples also show that view sets recommended by DFS are generally more efficient than those found by GSTR whether implicit triples are taken into account or not.

Overall, pre-computed views are likely to speed up query evaluation in any platform, simply by avoiding computations at runtime. Moreover, our framework (i) avoids the overhead of query rewriting at run-time, as query rewritings are also pre-computed, and (ii) could easily translate our rewritings directly to any RDF platform's logical plans, exploiting its physical optimization capabilities.

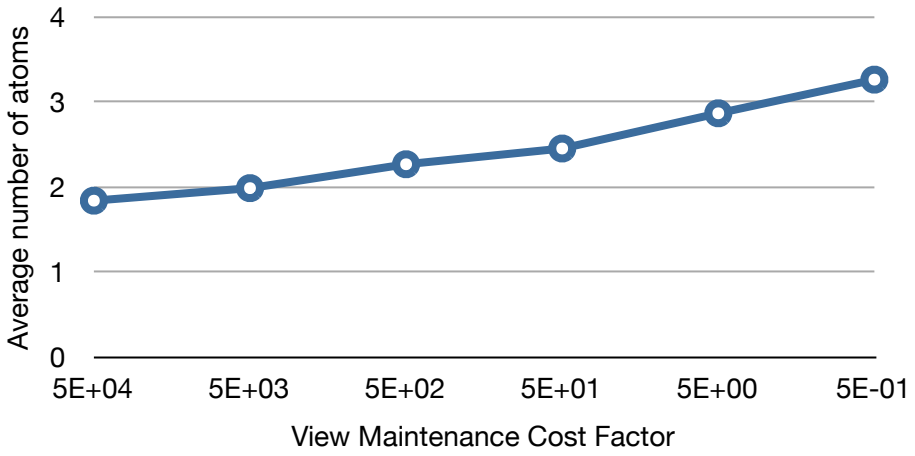


Figure 3.16: Average number of atoms per view in selected states.

3.7.7 Influence of the cost function components

To examine how each component of the cost function (see Section 3.3.4) affects the search, we ran several searches using DFS-AVF-STV with multiple combinations of the cost components. Since most of the cost reduction is achieved within the first minutes of search, we set the timeout at 5 minutes. Each workload had an average of 5 atoms per query, while workload sizes varied from 5 to 200 queries. Figure 3.15 shows the cost of the best state over time for typical workloads of 20 chain and mixed queries. Our online experiment page [www11] hosts a complete set of results with varying weights for the cost components. Overall, we observed that a reduction of at least 1 order of magnitude is achieved within 5 minutes regardless of the input workload size or type.

We now study the impact of c_m (the weight of the view maintenance component) on the characteristics of the proposed views and, in particular, on the average number of atoms in the views. We set $c_s = 1$ and $c_r = 1$ and varied c_m from $5 \cdot 10^4$ to 0.5. The results are depicted in Figure 3.16, showing that relaxing c_m has a direct impact on the number of atoms in the views. As expected, a strong VMC weight tends to promote smaller views of approximately 1.8 atoms on average (since the smaller the number of atoms, the smaller the maintenance cost).

3.7.8 Experiments conclusion

Our experiments have shown that the GSTR and DFS strategies scale well on up to 200 queries and achieve impressive cost reduction factors in many cases close to 99%. The strategies of [TLS01] are also effective for small workloads, but for larger ones they outgrow the memory before producing a solution. The AVF and STV heuristics are efficient and effective, i.e., they reduce the search space while preserving view set quality. Post-reformulation largely outperforms pre-reformulation in terms of speed and effectiveness of the proposed candidate view set. Finally, our recommended views reduce query evaluation times by several orders of magnitude. A digest of our experimental results can

found at [www11].

A tighter integration of the view selection tool with the internals of the data management platform, and/or using a dedicated RDF system, is likely to increase performance gains even more.

3.8 Related works

Our work is among the first to explore materialized view selection in RDF databases. The closest works related to ours are [CL10] and [DCDK11]. RDFMatView [CL10] recommends RDF indices to materialize for a given workload, while in [DCDK11] a set of path expressions appearing in the given workload is selected to be materialized, both aiming at improving the performance of query evaluation. Unlike our approach, none of these works aims at rewriting the queries completely using the materialized indices or paths and, thus, cannot be used in scenarios where the client needs to process her queries even without access to the database. Moreover, they do not consider the implicit triples that are inherent to RDF.

Many research works have addressed the efficient processing of RDF queries and updates, e.g., [AMMH07, SGK+08, WKB08, NW08, NW09, NW10b, UPS07], proposing various storage and indexing models. A detailed description of the existing RDF data management platforms is given in Chapter 2 of this thesis (Section 2.2.3). These techniques have been shown to result in good RDF query and update performance. We view our approach as complementary to these works, since we seek to identify materialized views to store *on top (independently) of* the base store and indexes. To adapt our approach to a specific RDF data management platform, one only needs (i) an execution framework capable of evaluating our simple select-project-join rewritings, and (ii) possibly tailoring the cost function to the particularities of the platform. Our approach improves performance by *exploiting pre-computed results and thus avoiding computations at query evaluation time*, gains likely to extend to any context.

Techniques to estimate the selectivity of RDF query patterns were proposed in [MASS08, SSB+08, NM11]. We compute the simple cardinalities advocated in RDF-3X [NW09], which are also shown to lead to satisfactory join size estimation. Other cardinality estimation frameworks could easily be plugged in our approach.

Materialized view selection has been intensely studied in relational databases [CHS02] and data warehouses [JLVV01]. We used [TLS01] as a starting point for our work, as it is one of the prevalent works in the area and the closest to our problem definition and query language. However, in [TLS01] the restriction that *no relation may appear twice in a workload query* is imposed, under which view equivalence can be tested in PTIME. This simplification is incompatible with RDF queries, which repeatedly use the triple table. In our context, determining view equivalence (needed for VF and for the search strategies) is NP-complete [CM77]. This, along with the typically bigger size of RDF queries compared to the relational ones (since only one table with three attributes is used), increase the complexity of the problem even more. Hence, the strategies presented in [TLS01] are not effective in our context. We innovate over [TLS01] by proposing new search strategies and heuristics, which, as demonstrated in Section 3.7, do not suffer from memory limita-

tions and lead to the selection of efficient views, even if we limit the time of the search. The set of transitions used in [TLS01] comprised: edge removal (ER′), attribute removal (AR′), view break (VB′), view merging (VM′), and attribute transfer (AT′). We modified them for our context as follows. First, AR′ and AT′ do not apply in our setting due to the differences between the SQL-like language they use and our Datalog formalism. In particular, AR′ considers that a given attribute (variable in our setting) can appear more than once in the query head, while AT′ assumes that constants may appear in the query head. Neither is supported in our Datalog formalism (we could extend it to include them, but this would not enlarge the set of candidate view sets). Second, in [TLS01], join edges are removed by ER′ (which may introduce a Cartesian product) but only VB′ can split a view in two smaller ones. We allow JC, which removes join edges, to also split a view in two if its graph has become disconnected. Finally, the transition VM′ of [TLS01] fuses views with inequality predicates, which are not needed in our RDF context.

More details about existing view selection approaches are provided in Chapter 2. In the same Chapter, the related problems of multi-query optimization (which identifies common query subexpressions) and view-based query rewriting (in which views are the input and not the output to the problem) are also discussed.

Query reformulation (also known as unfolding) is directly related to query answering under constraints interpreted in an open-world assumption (e.g., [Ros11]), i.e., when constraints are used as deductive rules. In particular, our query reformulation algorithm builds on those in the literature considering the so-called *Description Logic (DL) fragment* of RDF [AGR07, CGL$^+$07], i.e., description logic constraints. This fragment corresponds to RDF databases without blank nodes that are made of an RDFS, called a Tbox, and a dataset made of assertions for classes and properties in the RDFS, called an Abox, i.e., well-formed triples of the form $(\mathtt{s}, \mathrm{rdf{:}type}, c)$ or $(\mathtt{s}, p, \mathtt{o})$, where c is a class and p a property of the RDFS. Lastly, the RDF entailment rules considered are only those dedicated to an RDFS (see Section 3.4.1). Reformulation algorithms for the DL fragment of RDF actually reformulate queries from a strictly less expressive language than our RDF queries. They only support atoms in which the class or the property is specified i.e., they do not support atoms like $t(\mathtt{s}, \mathrm{rdf{:}type}, X)$ or $t(\mathtt{s}, X, \mathtt{o})$ with X a variable, stating respectively that $\mathtt{s}$ is an instance of a class or that $\mathtt{s}$ is somehow related to $\mathtt{o}$. To overcome this, our reformulation algorithm extends the state of the art to our RDF queries, i.e., the BGP of SPARQL.

3.9 Conclusion and future work

We considered the setting of a Semantic Web database, including both explicit data encoded in RDF triples, and implicit data, derived from the RDF entailment rules [www04a]. Implicit data is important since correctly evaluating a query against an RDF database also requires taking it into account. In this context, we have addressed the problem of efficiently recommending a set of views to materialize, minimizing a combination of query evaluation, view storage and view maintenance costs. Starting from an existing relational approach, we have proposed new search algorithms and shown that they scale to large query workloads, for which previous search algorithms fail. Notice

that our search algorithms can be applied not only in the context of RDF data, but also in the relational setting.

In the presence of an RDF Schema, our view selection approach can be used both with a saturated RDF database (where all implicit triples are added explicitly to the data), and with a non-saturated one (when queries need to be reformulated to reflect implicit triples). We have proposed a new algorithm for reformulating queries based on an RDF Schema, as well as a novel post-reformulation method for taking into account implicit triples in a query reformulation context. Post-reformulation can be much more efficient than naïve pre-reformulation, due to the high complexity of view search in the number of queries.

As future work, we consider parallelizing our view search algorithms by identifying workload queries that do not have many commonalities and running the search in parallel for each group. We also consider extending our query and view language, as well as adapting our approach to dynamic query workloads.

Chapter 4

Efficient XQuery Rewriting using Multiple Views

In this Chapter, we consider the problem of rewriting XQuery queries using multiple materialized XQuery views. The XQuery dialect we use to express views and queries corresponds to tree patterns (returning data from several nodes, at different granularities, ranging from node identifiers to full XML subtrees) with value joins. We provide the first sound and complete query rewriting algorithm for this problem. Our algorithm only finds minimal rewritings, that is, those in which no view is redundant. Our work extends the state of the art by considering more flexible views than the XPath 1.0 dialects previously considered, and more powerful rewritings.

The algorithm presented in this Chapter is a significantly expanded and reworked version of [MKVZ11]. We detail the novel content of this Chapter with respect to that publication in the following Section.

4.1 Motivation and outline

Query rewriting based on materialized views is a well-known performance enhancement technique in databases. While it was mostly used in relational databases [LRO96, PL00], query rewriting has recently received attention also in the context of XML databases, e.g., [ABMP07, BOB$^+$04, CDO08, TYÖ$^+$08, XO05].

In this Chapter, we study the problem of rewriting queries from an expressive XQuery dialect, using *multiple views* from the same dialect. As in [ABMP07, BOB$^+$04, CC10, CDO08, ODPC06, PZIÖ06, TYÖ$^+$08], we consider *equivalent rewritings*, which compute the same results as the original query, but *rely exclusively on the materialized views* (without accessing the original XML documents). In this context, given a set of views and a query, the task of evaluating the query can be split into three successive steps: (i) filter the view set to eliminate (as much as possible) those views which cannot be part of an equivalent query rewriting; (ii) find one or several rewritings of the query; (iii) through a process of logical and physical optimization, pick the rewriting which looks most promising, enumerate physical plans which may compute this rewriting, pick the best one and

evaluate it.

Task (ii) above, i.e., query rewriting, has generally high computational complexity even for simple view and query languages. This is why step (i) is important: any reduction in the number of views to use during rewriting is likely to significantly impact the time and memory needs of the rewriting. In the literature, view filtering is typically performed by testing some form of embedding from each view into the query [BOB$^+$04, MS05]. More recently, [TYÖ$^+$08] has proposed a highly efficient view filtering method, when the query and the views are expressed in XPath$^{\{/,//,*,[]\}}$. The filtering method is shown to perform well in practice, however, when it comes to step (ii), to avoid the high complexity of embedding tests in the presence of $*$ [MS04], the authors adopt some heuristics which in some cases make their proposed rewriting algorithm incomplete. Restricted to XPath$^{\{/,//,[]\}}$, the approach in [TYÖ$^+$08] is complete and very efficient. Optimizing and executing a rewriting (task (iii)) is a problem in itself, whose solution needs to take into account parameters such as the available storage structure and access methods, including possible indices on the materialized views, available physical operators, data distribution etc. Parameters characterizing a solution to this problem may be a cost model or an optimization strategy. Some recent works [CC10, PZIÖ06] focus on task (iii), providing efficient physical operators for view joins.

The focus in this work is on the core task (ii) above, that is: given a query and a set of views (which we assume already filtered), find all the possible equivalent rewritings of the query based on the views. Three main dimensions set our work apart from previous related works [BOB$^+$04, CC10, CDO08, ODPC06, PZIÖ06, TYÖ$^+$08].

First, we are only interested in *minimal* rewritings, i.e., those from which no view can be removed while still preserving the equivalence between the rewriting and the original query. We focus on minimal rewritings since regardless of the particular rewriting evaluation engine, a non-minimal rewriting will always entail more processing than a corresponding minimal one. Indeed, as the experiments we presented in [MKVZ11] show, the processing time difference between executing minimal and non-minimal rewritings is often very significant. To develop minimal rewritings only, we introduce a novel *bottom-up rewriting approach*, which builds partial rewritings by combining at every step, a smaller rewriting with an extra view. The combination either produces a rewriting over a bigger view set, or fails if it is non-minimal, i.e., if the new view was redundant with respect to the smaller rewriting. Thus, rewriting minimality is enforced and preserved throughout the process.

Second, we consider a rich dialect of XQuery, where a single query (respectively, view) can return (respectively, store) information from *several nodes*, and including *value joins*. In contrast, views in [BOB$^+$04, CDO08, TYÖ$^+$08] have a single return node, and each view must store: the subtree rooted at its return node, the node identifier (or ID, in short), and (in [BOB$^+$04]) the full label path to the node and other information. In our work, one can specify at various granularity levels what a view stores from each node, which again leads to more flexibility. Views and queries in [ODPC06] support group-by but not node IDs, which removes some rewriting opportunities while creating new ones. The focus in [CC10] and [PZIÖ06] is on task (iii), providing efficient physical operators in the particular case where views store IDs for *all* view nodes.

Finally, our rewritings are expressed in a *generic logical XML algebra*, compatible with many well-established XQuery processing platforms (see, e.g., [GJÖY09]). If the views are materialized as XML documents, our rewritings can directly be translated to XQuery statements, to be evaluated over the views by an off-the-shelf XQuery engine.

The contributions of this Chapter can be listed as follows.

1. We identify a joined tree pattern language corresponding to a rich, meaningful subset of XQuery. This language goes beyond the XPath 1.0 dialects considered in previous rewriting works [BOB$^+$04, CDO08, TYÖ$^+$08, XO05] in three important ways: (i) a single tree pattern may return data from several different nodes (thus, one tree pattern may return the year *and* the title of a publication), (ii) our language allows to distinguish between the identifier of a node, its text value, and its serialized image, in the spirit of the W3C XPath and XQuery data model, and (iii) we allow value joins in the language. Extensions (i) and (ii) have been previously made in the richer XAM tree pattern language [ABM05]. In the present work, we omit some XAM features (notably, nested and optional edges, as well as binding patterns) while adding value joins which are useful in many practical applications. Observe that some previous works on tree patterns, e.g., [MS04], also allowed $*$-labeled nodes, leading to the XPath$^{\{/,//,[]\}}$ dialect. The extension of our algorithm to $*$ nodes is left for future work.
   An important original contribution of the present Chapter is to define *containment* and *minimality* for the rich pattern considered in this work; we provide practical algorithms for deciding containment, and for minimizing such tree patterns. These questions have not been addressed in previous works.
2. We study the problem of rewriting queries expressed by means of such joined tree patterns, using *multiple* views expressed in the same formalism. This amounts to rewriting XQuery queries corresponding to this dialect, using similar XQuery views. We consider equivalent rewritings only, restrict our search to minimal rewritings, and provide a complete algorithm for solving this rewriting problem. To make search as efficient as possible, we identify *left-deep query tree-organized rewritings* (LDQTs, in short), a tight subset of all the possible minimal rewritings, and focus only on finding LDQTs. We show that any other minimal equivalent rewriting can be obtained from a corresponding LDQT by the optimizer.
3. An important component of our rewriting algorithm is the ability to manipulate a pattern by means of an algebra, in order to modify it in a specific desired way (add or remove a node, alter its structure, etc.). It turns out that due to fundamental differences between the two models (patterns versus algebras), the algebraic transformations needed to perform such transformations may be simple, complex, or they may not exist. To the best of our knowledge, ours is the first study of the *impact of algebraic transformations on a tree pattern*. This question is of interest in any XML processing context where queries correspond to some (tree) pattern formalisms, while the processing is modeled by algebraic operators.

Our rewriting algorithm has very high complexity in the total size of the query and views, which is expected since our work extends the language of [CDO08] (which estab-

lished coNP-hardness for XPath$^{\{/,//,[]\}}$ query rewriting using views with IDs) to a richer query and view language. For moderate-size queries and views and large databases, the high rewriting costs are still an acceptable price to pay, considering the reduction in query processing time that views may bring. This has been demonstrated by experiments we described in [MKVZ11].

This Chapter is a significantly enlarged and re-worked version of our publication [MKVZ11]. Contribution 1 above is completely new, while in this thesis we have also significantly re-structured the rewriting algorithm and studied the algebraic pattern transformations (contribution 3) that [MKVZ11] only hinted at, through examples. Note that in [MKVZ11], we had implemented an early version of the algorithm. The current version, described in this Chapter, is not fully implemented yet (it uses the same basic structure as in [MKVZ11], but includes many new building blocks).

The remainder of the Chapter is organized as follows. Section 4.2 outlines the rewriting problem we consider, and our solution to this problem, based on an example. The next Sections then detail our approach. Section 4.3 specifies the XQuery dialect we use for views, queries and rewritings, as well as the internal models used by our algorithm: tree patterns and algebra. Section 4.4 provides our study of important properties for our patterns, as well as for DAG-shaped patterns, not used in our views and queries, but produced during the rewriting process. For readability, we first describe our complete algorithm for building minimal rewritings in the restricted case when both the query and views consist of single tree patterns (no value joins across patterns) in Section 4.5. Section 4.6 discusses how one can modify a tree pattern by applying algebraic transformations. Based on these, Section 4.7 details the inner workings of the rewriting algorithm. Section 4.8 generalizes the algorithm from Section 4.5 to handle our general language (with value joins). We then provide more details on the related works, and we conclude.

4.2 Motivating Example

Our query rewriting approach is illustrated by the example depicted in Figure 4.1. The Figure shows two views v_1 and v_2, a query q, and three increasingly larger logical plans p_1, p_2, and p_3, such that p_3 is an equivalent rewriting of q using v_1 and v_2. The Figure shows the query and views as tree patterns, and the plans in an algebraic formalism. The details on their corresponding XQuery syntax and on the algebra will be provided in Section 4.3.

Each solid single (double) edge in Figure 4.1 denotes parent-child (ancestor-descendant) relationships. The view v_1 stores the identifiers (or IDs, in short) and the contents (the full XML subtrees, without including the IDs of the corresponding nodes, denoted $cont$) of the affiliations of all conference papers, along with the IDs of these papers. View v_2 stores the title and the publishing country of the books edited by IEEE and published the same year as an ICDE paper whose author must have an email, as well as the IDs of these papers and of their authors. We use the notation val to denote the string value of an element, obtained by concatenating all the text descendants of that element [w3c07d]. The predicate $[=IEEE]$ imposes that the val of the respective element should be equal to 'IEEE'. The dashed line connecting the two nodes labeled year in v_2 joins the two tree patterns, on the condition that the value of the year elements be the same on both sides.

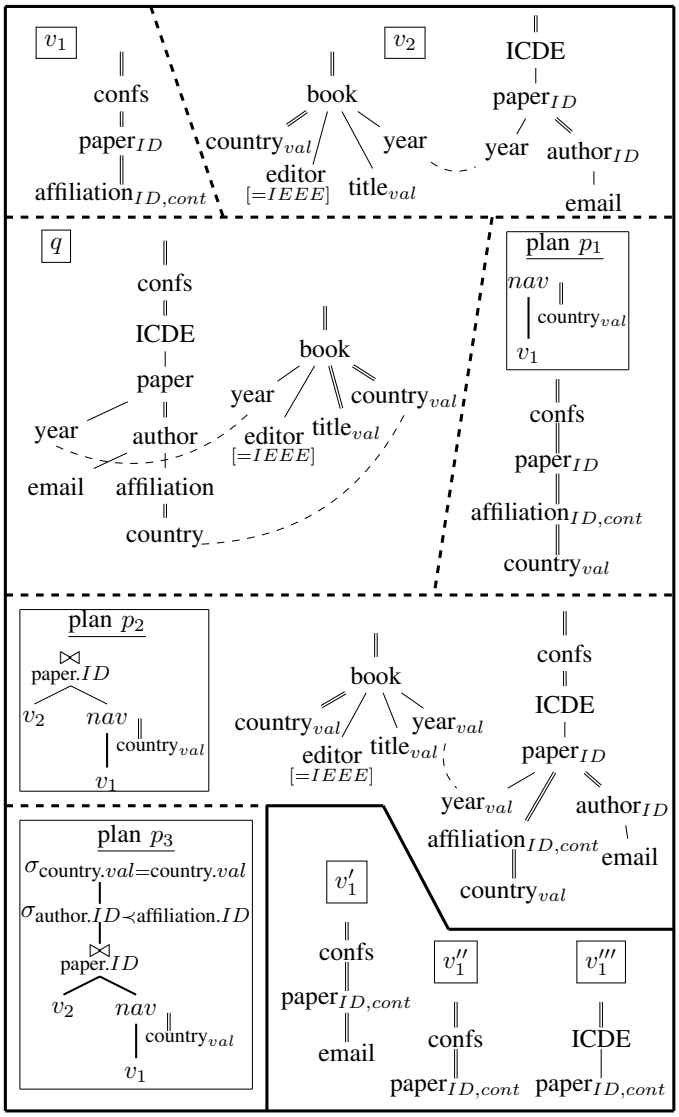


Figure 4.1: Sample views, query, and algebraic rewritings.

The query q asks for the title and country of books edited by IEEE and published the same year and in the same country as some author of an ICDE paper, for which an e-mail is specified.

We now show how we build an equivalent algebraic rewriting of q using v_1 and v_2. The query asks for the country in author/affiliation. Plan p_1 applies a *navigation* operator, denoted nav, on top of the view v_1, searching for descendants labeled country within the affiliation subtrees, and retaining their string values. Here, the small tree pattern //country$_{val}$ is a parameter of the algebraic nav operator. Underneath p_1, we show the tree pattern equivalent to this plan. Navigation has added the bottom country$_{val}$ node.

Plan p_2 joins the view v_2 with the plan p_1, on the paper ID. At its right, we show the pattern equivalent to this plan. Note that the paper node has now three children (year,

author, and affiliation). Observe that after the join on paper ID, the parent of the paper node is labeled ICDE (as in v_2), whereas the confs node is pushed one level up, as an ancestor of the ICDE node. This is because the paper node can only have one parent, and v_2 specifies its label is ICDE. The confs node must then be an ancestor of the ICDE node in the join result. Such reasoning has first been made in [CDO08], for a strict subset of our language (with only one return node and no value joins). In [TYÖ+08], when joining two XPath views on their target node ID, the order among the ancestors of the join node can be established by a *physical join operator* over the views, exploiting the expressive IDs they use. The join may return an empty result if the nodes belong to different paths, and this would only be detected at runtime. In our work, as in [CDO08], we do such reasoning *statically* on the views, without accessing the view data.

We now discuss how to go from p_2 to the equivalent rewriting p_3. A first important remark is that in the pattern produced by the plan p_2, affiliation is a sibling of author, instead of being its child. Therefore, an *adaptation* is needed, materialized by the lower selection in p_3, namely $\sigma_{\text{author.ID}\prec\text{affiliation.ID}}$, where the $\prec$ symbol stands for a binary "isParentOf" predicate. We assume in this example that the IDs of author and affiliation are *structural*, that is, one can determine the structural relationships (parent or ancestor) between two nodes just by comparing their IDs. If the IDs were not structural, no equivalent rewriting of q using v_1, v_2 exists, since one cannot ensure the affiliation nodes from v_1 are children of the author nodes from v_2. (In cases not requiring structural predicates, we may obtain rewritings even based on simple IDs.)

Our rewriting algorithm then realizes that the query-specified join on year is already applied by v_2, whereas equality of the two country nodes still needs to be enforced. The rewriting is completed by applying the top selection of plan p_3 (and a final projection, not shown in Figure 4.1).

We now illustrate the differences between our work and the closest related ones [CDO08, TYÖ+08] using XPath views. Since these works do not handle value joins, for the purpose of the comparison, we restrict q to its leftmost tree pattern only. Moreover, as the other approaches do not support queries and views with multiple return nodes, we have to further simplify the query so that only one node is returned, e.g. the val of country nodes. Among the views which [CDO08] and [TYÖ+08] may use to rewrite this restricted query are v_1', v_1'' and v_1''' shown at the bottom of Figure 4.1 (one could also copy under the paper nodes of v_1', v_1'' and v_1''' the subtrees of the paper query node, as existential branches). Observe that v_1', v_1'' and v_1''' store whole paper contents, which may be much larger than what the query needs. This drawback is due to the single return node in XPath 1.0, forcing the view to store the least common ancestor of all nodes from which some information is returned by the query. In contrast, v_1 and v_2 only store what the query needs. Moreover, [CDO08] and [TYÖ+08] would produce a rewriting using all of v_1', v_1'' and v_1''', whereas our algorithm understands that v_1'' and v_1''' suffice for a minimal rewriting. A procedure to minimize non-minimal rewritings is sketched in [TYÖ+08], however, it relies on the special properties of the IDs they use.

1	q :=	for $absVar$ (, ($absVar$|$relVar$))* (where $pred$ (and $pred$)*)? return ret
2	$absVar$:=	x_i in doc(uri) p
3	$relVar$:=	x_i in x_j p // x_j *introduced before* x_i
4	$pred$:=	string(x_i) = (string(x_j) | c)
5	ret :=	$\langle l \rangle$ $elem$* $\langle /l \rangle$
6	$elem$:=	$\langle l_i \rangle$\{ (x_k | id(x_k) | string(x_k)) \}$\langle /l_i \rangle$

Figure 4.2: Grammar for views and queries.

4.3 Views, Queries and Problem Statement

In this Section, we describe the XML query dialect we consider in Section 4.3.1. We then present a joined tree pattern formalism on which we will rely for our algorithm, in Section 4.3.2. Based on these, Section 4.3.3 formalizes the rewriting problem we address.

4.3.1 XQuery Dialect

Let $\mathcal{L}$ be a finite set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language [MS04]. We consider views and queries expressed in the XQuery dialect described in Figure 4.2. In the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the bindings of a previously-introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i, either with the string value of another variable x_j, or with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause: (i) x_k denotes the full subtree rooted at the binding of x_k; (ii) string(x_k) is the string value of the binding; (iii) id(x_k) denotes the ID of the node to which x_k is bound.

There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of n's text descendants [w3c07d]. Therefore, string(x_i) is very likely smaller than x_i's content, but it holds less information. Second, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones. Our XQuery dialect distinguishes IDs, value and contents, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

For illustration, Figure 4.3 shows the query q and the views v_1 and v_2 from Figure 4.1,

q	for $p in doc("confs")//confs//ICDE/paper, $y1 in $p/year,
	$a in $p//author[email], $c1 in $a/affiliation//country,
	$b in doc("books")//book, $y2 in $b/year, $e in $b/editor,
	$t in $b//title, $c2 in $b//country
	where $e='IEEE' and $y1=$y2 and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{string($t)}⟨/tval⟩ ⟨/res⟩

v_1	for $p in doc("confs")//confs//paper, $a in $p/affiliation return ⟨v1⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨acont⟩{$a}⟨/acont⟩ ⟨/v1⟩

v_2	for $b in doc("books")//book, $c in $b//country, $e in $b/editor,
	$t in $b/title, $y1 in $b/year, $p in doc("confs")//ICDE/paper,
	$y2 in $p/year, $a in $p//author[email]
	where $e='IEEE' and $y1=$y2
	return ⟨v2⟩ ⟨cval⟩{string($c)}⟨/cval⟩ ⟨tval⟩{string($t)}⟨/tval⟩
	⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨/v2⟩

r	for $v1 in doc("v1.xml")//v1, $p1 in $v1/pid, $af1 in $v1/aid,
	$c1 in $v1//acont//country, $v2 in doc("v2.xml")//v2,
	$c2 in $v2/cval, $t2 in $v2/tval, $p2 in $v2/pid, $a2 in $v2/aid
	where $p1=$p2 and parent($a2,$af1) and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{$v2/tval}⟨/tval⟩ ⟨/res⟩

Figure 4.3: Sample query, views, and rewriting.

in our XQuery dialect. The XQuery expression r corresponds to the algebraic plan p_3 of Figure 4.1, dictating how q can be answered using exclusively v_1 and v_2. The parent custom function returns true *iff* inputs are node IDs, such that the first identifies the parent of the second. Moreover, as usual in XQuery, the variable bindings that appear in the where clauses imply the string values of these bindings (e.g. $e='IEEE' is implicitly converted to string($e)='IEEE').

Order of results As per XQuery semantics, results follow the order of the bindings of the for variables, and are thus obtained in the order they appear in the query. For instance, in Figure 4.3, results of v_1 are ordered first by the $p bindings and then by $a bindings etc. Our rewriting preserves query semantics including such duplicates, and result order.

4.3.2 Patterns and Algebra

We use a dialect of joined tree patterns to internally represent views and queries, some examples of which appeared in Figure 4.1. Section 4.3.2.1 formally presents our joined pattern dialect, whose semantics we define next. Section 4.3.2.2 defines pattern semantics based on embeddings from individual tree patterns into XML documents. Given the important role that the XML algebra plays in our context, we also provide an alternative, equivalent definition of joined tree pattern semantics in Section 4.3.2.3, based on this algebra.

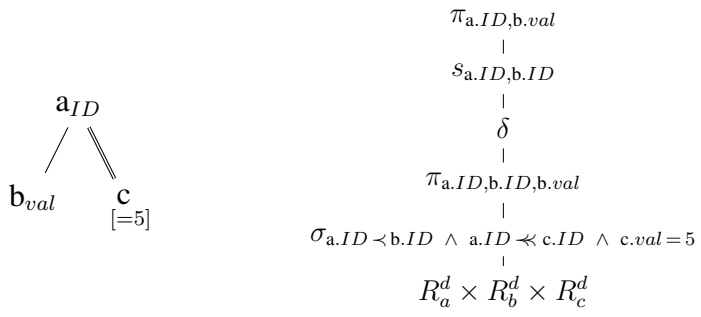


Figure 4.4: Sample tree pattern and its algebraic semantics.

4.3.2.1 Tree and Joined Patterns

Formally, a *tree pattern* p is a tree whose nodes carry labels from $\mathcal{L}$. Each node $n \in p$ may be annotated with zero or more among: ID, val and $cont$, standing for n's ID, the concatenation of the values of n's descendant text nodes, and the serialization of the full subtree rooted at n, respectively. Moreover, n may be annotated with a value equality predicate of the form $[=c]$ where c is some constant. The pattern edges are either simple for parent-child or double for ancestor-descendant relationships.

A *joined (tree) pattern* is a set of tree patterns, connected through value joins, which are denoted by dashed edges.

As can be seen comparing Figures 4.1 and 4.3, the translation from our XQuery dialect to the joined tree patterns is quite straightforward. The only XQuery syntax aspect not reflected in the joined tree patterns is the name assigned to elements created by the return clause. This information is irrelevant to rewriting, therefore it is directly transmitted to the optimizer which will insert the appropriate element constructor operators on top of the rewritings we produce.

4.3.2.2 Pattern Semantics Based on Embeddings

We now define the semantics of (joined) tree patterns based on *embeddings*. Note that embeddings have already been used to define semantics of other tree pattern dialects, e.g., in [AYCLS02].

Tree pattern semantics Let ϕ be an embedding from a tree pattern p to a document d. As customary [AYCLS02], ϕ preserves node labels, and if n is a $/$-parent (respectively, a $//$-parent) of node m in tp, then $\phi(n)$ is a parent (respectively, a parent or ancestor) of $\phi(m)$ in d. The semantics of p on d, denoted $p(d)$, is a list of tuples. The attributes in each tuple are obtained by traversing p in depth-first order and for each p node n, adding $n.ID$ if n is annotated with id, then $n.val$ if n is annotated with val, then $n.cont$ if n is annotated with $cont$. For instance, the semantics of the tree pattern p in the left-hand side of Figure 4.4 is a table with tuples of type (a.ID, b.val). There is a tuple in $p(d)$ for each embedding of p in a distinct (a,b) pair of d nodes. The tuples in $p(d)$ follow the order of their corresponding (a, b) nodes.

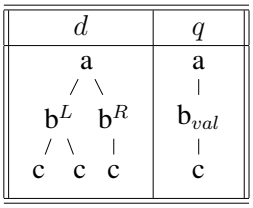


Figure 4.5: Set (1 tuple), bag (3 tuples) and XQuery (2 tuples) semantics.

Joined pattern semantics Let jp be a joined pattern and $\mathcal{D}$ a set of documents. There is an embedding ψ from jp to $\mathcal{D}$ if there is a tree embedding [AYCLS02] from each tree pattern of jp to a document in $\mathcal{D}$, and the join predicates of jp are respected by the values of the documents to which the tree patterns of jp are embedded. The semantics of $jp(\mathcal{D})$ is again a list of tuples, and the attributes in each tuple are obtained by concatenating the attributes appearing in the tuples of the tree patterns of jp. Each tuple corresponds to a distinct embedding of the return nodes of jp on the documents of $\mathcal{D}$.

Duplicates: bag, set and XQuery semantics When evaluating a joined pattern query q over a document d, depending on the considered semantics (bag, set or XQuery), the results of the evaluation may differ in terms of cardinality. In all cases, to compute $q(d)$, we first find all the embeddings of q on d. Then, under bag semantics, we directly output these embeddings; under set semantics, we perform a duplicate elimination on the results (we keep no duplicates); finally, under XQuery semantics, which is the one we target and was described above, we keep the distinct embeddings determined by the return nodes of q, but remove the duplicates brought by the existential nodes. As an example, let d and q be the document and query, respectively, depicted in Figure 4.5. Notice that q can be embedded 3 times in d, c is an existential node in q, and assume that $b^L.val = b^R.val$. Under bag semantics we would return 3 tuples, under set semantics 3 tuples, whereas under XQuery semantics 2 tuples (the double embedding on b^L is considered only once).

4.3.2.3 Algebraic Pattern Semantics

To better highlight the connection between views and rewritings, we provide an alternative algebraic definition of the semantics. This semantics is a restriction of the more general tree patterns semantics, defined in [ABM05], which we extend for the case of joined patterns.

Algebra Let $\mathcal{A}$ be the algebra consisting of the following operators:

1. scan of all tuples from a view v, denoted $scan(v)$ (or simply v for brevity, whenever possible);
2. selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are attribute names, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$;
3. cartesian product, denoted $\times$. We also use joins, defined, as usual, as selections over $\times$;

4. projection, denoted π_{cols}, where $cols$ is the attributes list that will be projected
5. duplicate elimination (denoted δ)
6. sort, denoted s_{cols}, where $cols$ is the list of attributes defining the ordering;
7. navigation, denoted $nav_{a,np}$, is a unary algebraic operator, parameterized by one of its input columns' name a, and a tree pattern np. The name a must correspond to a $cont$ attribute. Let t be a tuple in the input of nav, and $np(t.a)$ be the result of evaluating the pattern np on the XML fragment stored in $t.a$. Then, $nav_{a,np}$ outputs the tuples $\{t \times np(t.a)\}$, obtained by successively appending to t each of the tuples in $np(t.a)$.

An example of navigation appeared in the plan p_1 of Figure 4.1. The plan p_1 outputs 4-attribute tuples: paper IDs, affiliation IDs and contents (coming from v_1), as well as country values (by virtue of the navigation).

The nav operator adds attributes and may add tuples, all the while possibly removing other tuples. For instance, let p_1 denote the pattern $//a_{cont}$, let e_1 be an algebraic expression equivalent to p_1, and assume that on a document d, we have:

$p_1(d)$={("<a/>"), ("<a>b1b2b3")}

Then, $nav_{a_{cont},/b_{cont}}(e_1)(d)$, corresponding to the pattern $p_2 = //a_{cont}/b_{cont}$, is:

{("<a>b1b2b3", "b1"),
("<a>b1b2b3", "b2"),
("<a>b1b2b3", "b3")}

Observe that the first tuple has been erased by nav (since there was no match of the $/b$ pattern in this tuple), while the second tuple has been transformed into three, one for each b element.

Tree pattern semantics Given a document d and label $a \in \mathcal{L}$, we denote by R_a^d and call *virtual canonical relation of a in d*, the list of tuples of the form $(n.ID, n.val, n.cont)$ obtained from all the a-labeled nodes n in d. The tuples in R_a^d follow the order of appearance of the corresponding nodes in d. We denote by $\prec$ the *parent comparison operator*, which returns true if its left-hand argument is the ID of the parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the *ancestor comparison operator*. Observe that $\prec$ and $\prec\!\!\prec$ are only abstract operators here (we do not make any assumption on how they are evaluated).

We consider first the case when all pattern nodes are annotated with ID, val and $cont$, and no value predicate. Then, $p(d)$ is an algebraic expression obtained as follows:

1. Let R_p be the set of virtual canonical relations obtained by considering R_a^d once for each p node labeled a;
2. For each edge e from the p node n to a child m, let $pred(e)$ be the predicate $n.ID \prec m.ID$ (if e is a parent-child edge) or $n.ID \prec\!\!\prec m.ID$ (if e is an ancestor-descendant edge);

3. Define $p(d)$ as $s_{ID}(\sigma_{\bigwedge_e(pred(e))}(\times_{R_i \in R_p}(R_i)))$, where s_{ID} denotes sorting by the IDs of all tree pattern nodes, ordered by a left-deep traversal of the tree pattern.

We now consider predicates of the form [=c] on p nodes. Let $pred(n)$ be the value predicate of a tree pattern node n (or $true$ if n has no predicate). Then, $p(d)$ becomes:

$$s_{ID}(\sigma_{\bigwedge_e(pred(e)) \wedge \bigwedge_n(pred(n))}(\times_{R_i \in R_p}(R_i)))$$

We now consider the general case. Given a tree pattern p, let π_1^p be a projection operator which retains from any p node annotated with at least one among ID, val or $cont$: its ID attribute, whether or not n is annotated with ID, and its val (resp. $cont$) attribute, if the node is annotated with val (resp. $cont$).

Further, let π_2^p be a projection operator which retains from any p node, its ID (resp., val or $cont$), if the node is annotated with ID (resp., val or $cont$). Based on these ingredients, the semantics of a general tree pattern is defined as:

$$\pi_2^p(s_{ID}(\delta(\pi_1^p(\sigma_{\bigwedge_e(pred(e)) \wedge \bigwedge_n(pred(n))}(\times_{R_i \in R_p}(R_i))))))$$

Intuitively, the inner projection π_1^p followed by duplicate elimination turns to non-return all the nodes which do not project any attribute in the tree pattern. However, we need two projections, the inner keeping some IDs not required by the tree pattern (to preserve semantics during duplicate elimination and enable sorting), and the outer one, which keeps exactly the attributes required by the tree pattern.

As an example, Figure 4.4 depicts a tree pattern, along with the algebraic expression corresponding to its semantics.

Joined tree pattern semantics Let jp be a joined tree pattern over the set $\mathcal{T}_{jp} = \{p_1, p_2, \ldots, p_m\}$ of tree patterns. Let $\mathcal{VJ}_{jp} = \{vj_1, vj_2, \ldots, vj_k\}$ be the set of its value join predicates, where each vj_i is of the form $p_{i_1}.n_1.val = p_{i_2}.n_2.val$, imposing that the value of node n_1 from the tree pattern p_{i_1} be equal with that of the node node n_2 from the tree pattern t_{i_2}, $1 \leq i_1, i_2 \leq m$. Before defining the formal semantics of a joined pattern, we introduce its extended version.

Definition 4.3.1 (Extended joined pattern)**.** *Let jp be a joined pattern. The* extended version *of jp, denoted jp^x, is obtained from jp by adding a val attribute to every node of jp participating in a value join (if the node did not already have val).*

By definition, every node n and every tree pattern p in jp has a corresponding node n^x and tree p^x (which we call extended tree pattern), respectively, in jp^x.

The semantics of jp is defined by the algebraic expression $\pi_{jp}(\sigma_{\bigwedge_{i=1,\ldots,k} vj_i}(p_1^x \times p_2^x \times \ldots \times p_m^x))$, where π_{jp} is a projection retaining all the attributes of the original patterns p_1, p_2, $\ldots, p_m$, and eliminating the val attributes not present in $p_1, p_2, \ldots, p_m$. Note that while each tuple in the semantics of a *tree* pattern corresponds to data from a single document, a tuple in a joined tree pattern may be built with data from different documents.

4.3.3 Formal Problem Statement

We first formalize equivalent and minimal rewritings.

Definition 4.3.2 (Equivalent rewriting)**.** *Given a set of joined pattern views $\mathcal{V}$ and a joined pattern query q, an equivalent rewriting (or rewriting, in short) of q using $\mathcal{V}$ is an $\mathcal{A}$ expression e whose leaves are views from $\mathcal{V}$, such that for any document d, $e(d) = q(d)$, that is, evaluating e over d yields the same results as evaluating q over d.*

Definition 4.3.3 (Minimal rewriting)**.** *A rewriting e of the query q using $\mathcal{V}$ is minimal if no other rewriting of q uses a proper subset of the view instances used in e.*

Minimal rewritings should be distinguished from *min-size* rewritings, i.e., the (minimal) rewritings using the smallest possible number of views. Min-size rewritings do not always lead to the most efficient plans. For instance, the single view ($//\text{conf}_{ID,cont} \times //\text{book}_{ID,cont}$) suffices to answer the query q in Figure 4.1, yet it is very large and the rewriting based on v_1 and v_2 is likely to be much more efficient.

Intuitively, minimal rewritings are likely to lead to more efficient evaluation plans than non-minimal ones. In particular, under some generic assumptions on the rewriting evaluation cost model, it can be shown that a minimal is guaranteed to have lower cost than any non-minimal rewriting built on it. More specifically, we assume that:

- The cost to scan a view is linear in the size of the view and the scan cost per tuple is the same for all views.
- The cost of a join is linear in the size of each input and in the size of the output. This is the case, for instance, for hash-joins, merge-joins and XML specific join algorithms [AKJP+02].

Observe that in some setting these assumptions may be too restrictive, e.g., when data transfers are involved, when indexes are available on some of the views.

Therefore, the problem we consider is: *given a query q and a view set $\mathcal{V}$, find minimal $\mathcal{A}$ rewritings of q using $\mathcal{V}$*. The best minimal rewriting should be chosen by the optimizer.

4.4 Preliminaries on Patterns

Our query rewriting approach requires different kinds of reasoning on queries expressed in the joined tree pattern language we consider. This language is a restricted conjunctive version of XAMs [ABM05], and can be viewed as (conjunctive) tree patterns with multiple return nodes and many possible return attributes, connected by value joins. Fundamental notions such as the containment, equivalence and minimality for our joined tree pattern language have not been defined in previous works, and they turn out to be quite different from the ones which apply in the more traditional setting of single return node XPath patterns [AYCLS02, MS04].

In this Section, we define such fundamental notions and provide efficient procedures to test when they hold. For readability, we first consider the restriction of our language to single tree patterns (with multiple return node and possibly multiple attributes per return node). Section 4.4.1 defines containment among such tree pattern queries, and

provides a PTIME procedure for checking containment among such tree patterns. Section 4.4.2 defines the notion of tree pattern minimality in our context, and shows that an existing minimization algorithm can be slightly modified to also apply in our case. Section 4.4.3 introduces tree pattern equivalence and provides a Theorem which leads to a polynomial-time equivalence test. Next, Section 4.4.4 revisits the notions of containment, equivalence and minimization for the general case of joined tree pattern queries. Finally, Section 4.4.5 studies containment among DAG patterns, which generalize our (many return nodes, many attributes) tree patterns by allowing nodes to be organized in a directed acyclic graph, not necessarily a tree. The reason we consider DAG patterns is that, although the views and queries we consider only feature (possibly joined) tree patterns, we may obtain DAG patterns as a result of applying algebraic operations on our views. The connection between many-views XML rewriting and DAG patterns has also been made, e.g., in [CDO08]. We revisit it here for our context where multiple nodes may return various subsets of attributes.

4.4.1 Tree Pattern Containment

A crucial notion for the rewriting problem we consider, is that of tree pattern containment. To define it for our specific brand of tree patterns, we start with an auxiliary notion:

Definition 4.4.1 (Tree pattern signature)**.** *Let p be a tree pattern and $\{a_1, a_2, \ldots, a_m\}$ the set of return attributes of p. For each attribute $a_i, 1 \leq i \leq m$, let l_i be the label of the node that attribute a_i is attached to and $t_i \in \{ID, cont, val\}$ the type of a_i. The* signature *of p, denoted $S_{at}(p)$, is the multiset $\{l_1.t_1, l_2.t_2, \ldots, l_m.t_m\}$.*

We also use the notion of *node signature* for a node n of a tree pattern p, denoted $S_{at}(n, p)$, defined as the subset of $S_{at}(p)$ referring only to the attributes of n. For example, in Figure 4.6, $S_{at}(p_1) = \{b.val, e.ID\}$ and $S_{at}(p_2) = S_{at}(p) = \{b.val, b.val\}$. Moreover, $S_{at}(e, p_1) = \{e.ID\}$.

We are now ready to define tree pattern containment:

Definition 4.4.2 (Tree pattern containment)**.** *Let p, p' be two tree patterns. Pattern p is contained in p', denoted $p \sqsubseteq p'$, if (i) $S_{at}(p) = S_{at}(p')$, and (ii) for any document d, $p(d) \subseteq p'(d)$.*

Observe that in our setting, containment only holds for patterns having the same signature. For instance, in Figure 4.6, p and p_1 are incomparable since p returns tuples of the form $(b.val, b.val)$, while p_1 returns tuples of the form $(b.val, e.ID)$.

Our next task is to identify criteria for deciding when a pattern is contained in another. To that end, we define:

Definition 4.4.3 (Structural embedding)**.** *Given two tree patterns p and p', a* structural embedding *ϕ from p' to p, denoted $\phi : p' \xrightarrow{s} p$, is a function that maps each node in p' to some node in p, respecting node labels and edge relationships:*

1. *if r is the root of p', then $\phi(r)$ is the root of p;*

d	p	p_1 ($p \not\sqsubseteq p_1$)	p_2 ($p \not\sqsubseteq p_2$)	p_3 ($p \not\sqsubseteq p_3$)	p_4 ($p \sqsubseteq p_4$)
a / \ b^L e | | f b^R	a / \ b_{val} e | | f b_{val}	a / \ b_{val} e_{ID}	a / \ b_{val} b_{val} | f	a / ∖ b_{val} b_{val} | $f_{[=5]}$	a / ∖ b_{val} b_{val} | f

d'	p'_1	p'_2 ($p'_1 \not\sqsubseteq p'_2$)	p'_3	p'_4 ($p'_3 \sqsubseteq p'_4$)
a / \ $b^{L'}$ $b^{R'}$	a / \ b_{val} $b_{ID,cont}$	a / \ $b_{ID,val}$ b_{cont}	a | $b_{ID,val}$	a / \ b_{ID} b_{val}

Figure 4.6: Tree pattern containment.

2. *for each node n of p', n and $\phi(n)$ have the same label;*
3. *for each $/$-edge (n_1, n_2) in p', $(\phi(n_1), \phi(n_2))$ is a $/$-edge in p;*
4. *for each $//$-edge (n_1, n_2) in p', there is a path from $\phi(n_1)$ to $\phi(n_2)$ in p.*

In Figure 4.6, there is a structural embedding from each of the patterns p_1, p_2, p_3 to p, as well as from p'_2 to p'_1, and from p'_4 to p'_3.

For tree patterns belonging to the XPath$^{\{/,//,[]\}}$ dialect, it has been shown in [AYCLS02] that the existence of a structural embedding from p' to p is a necessary and sufficient condition for p to be contained in p' (i.e., $p \sqsubseteq p'$). The same holds for the larger XPath$^{\{*,/,//,[]\}}$ dialect [MS04]. Interestingly, their findings also hold for tree patterns with multiple return nodes, but this only corresponds, in terms of our tree pattern language, to patterns such that any return node can only return ID.

In contrast, in our setting, a structural embedding, although necessary, is not a sufficient condition for containment. To see why, let $Mod(p)$ be the set of documents d, for which $p(d)$ is non-empty. For boolean and single return node tree patterns, for $p \sqsubseteq p'$ to hold, it suffices to show that $Mod(p) \subseteq Mod(p')$; in turn, this inclusion is guaranteed by $p' \xrightarrow{s} p$ [MS04]. For our patterns with multiple return nodes, for $p \sqsubseteq p'$ to hold, we also need that $Mod(p) \subseteq Mod(p')$, but this is insufficient. For instance, consider the document d shown in Figure 4.6, where for the purpose of the explanation, we have distinguished the two b elements as b^L at the left and b^R at the right. In this Figure, $p \not\sqsubseteq p_2$: indeed, the tuple (b^L_{val}, b^R_{val}) belongs to $p(d)$, while it does not belong to $p_2(d)$. Thus, containment does not hold, even though clearly $Mod(p) \subseteq Mod(p_2)$. Likewise, in the same Figure, $Mod(p'_1) = Mod(p'_2)$, however $p'_1 \not\sqsubseteq p'_2$, as witnessed by the document d'. As soon as the elements $b^{L'}$ and $b^{R'}$ from d' do not have identical values, the tuple ($b^{L'}_{val}$, $b^{R'}_{ID}$, $b^{R'}_{cont}$) belongs to $p'_1(d')$ and it does not belong to $p'_2(d')$. The intuition behind these examples is: for a pattern p to be contained in a pattern p', both with multiple return nodes, the attributes of the return nodes of p' should be able to take values from all the document node sets that the attributes of p can take values.

A final observation concerns the predicates which may be found in patterns. For instance, in Figure 4.6, $p \not\sqsubseteq p_3$, because p_3 has a predicate that is not present in p, making p_3 more restrictive than p.

We formalize this discussion by the following criterium for deciding containment of tree patterns with multiple return nodes:

Theorem 4.4.1 (Tree pattern containment)**.** *Let p, p' be two tree patterns. $p \sqsubseteq p'$ iff:*

1. $S_{at}(p) = S_{at}(p')$*;*
2. *there exists an embedding $\phi : p' \xrightarrow{s} p$ such that:*
   (a) *for each node n' of p', $S_{at}(n', p') \subseteq S_{at}(\phi(n'), p)$;*
   (b) *if n_1, n_2 are p' nodes such that $\phi(n_1) = \phi(n_2)$, then $S_{at}(n_1, p') \cap S_{at}(n_2, p') = \emptyset$;*
   (c) *if the node n' of p' has a value predicate $[= c]$, then $\phi(n')$ in p has the same predicate.*

Proof. ("If" direction) Assuming the above hypotheses hold, we show that for any document d and for any tuple $t \in p(d)$, we have $t \in p'(d)$. Since $t \in p(d)$, there exists an embedding ψ_t from p to d, mapping the return nodes of p to a set of nodes in d. Note that each attribute a_n of a return node $n \in p$ is evaluated on node $\psi_t(n) \in d$. One can compose ψ_t with the embedding ϕ provided by the hypothesis *2*, and obtain $\psi'_t = \psi_t \circ \phi$, which is an embedding from p' into d. Our task now is to show that ψ'_t leads to the same tuple t being present in $p'(d)$. More specifically, we show that for any return node $n \in p$ having the attribute a_n, where $a_n \in \{ID, val, cont\}$, there exists a node $n' \in p'$ such that $\phi(n') = n$ and n' is also annotated a_n. This way, ψ'_t will give to the return attributes of p' the same values that ψ_t gives to the attributes of p, and thus have $t \in p'(d)$.

Assume to the contrary that for some node $n \in p$, such an n' does not exist. Still, there must exist some $n'' \in p'$ having the attribute a_n, otherwise $S_{at}(p) \neq S_{at}(p')$, which contradicts hypothesis *1*. Let n_p be the node of p such that $\phi(n'') = n_p$; by construction, $n_p \neq n$. If n_p does not have an a_n attribute, then $S_{at}(n'', p') \not\subseteq S_{at}(\phi(n''), p)$, which contradicts our hypothesis *2(a)*. If n_p has such an attribute, for $S_{at}(p) = S_{at}(p')$ to hold, there should be another node $n_{p'} \in p'$ having a_n, and such that $\phi(n_{p'}) = n_p$. This contradicts hypothesis *2(b)*, which dictates, since n'' and $n_{p'}$ both embed into n_p, that $S_{at}(n'', p) \cap S_{at}(n_{p'}, p) = \emptyset$.

These contradictions show that $n' \in p'$ must exist as we required. Thus, whenever ψ_t gives values to the attributes of p from a set of d nodes, $\psi_{t'}$ also gives the same values to the attributes of p'. Moreover, hypothesis *2(c)* ensures that p' is less restrictive than p in terms of value predicates. All these guarantee that $t \in p'(d)$.

("Only if" direction) Obviously, condition *1* holds by Definition 4.4.2. We show that if any other hypothesis does not hold, then $p \not\sqsubseteq p'$, i.e., for some document d there exists a tuple $t \in p(d)$ such that $t \notin p'(d)$.

We build a document d based on p as follows. For every node in p we add a node in d with the same label, and for each edge in p, we add a parent-child edge in d. Moreover, every value predicate on a p node is satisfied by the value of the corresponding d node. Let ψ_d be the embedding from p to d. Obviously p' can also be embedded to d through the composition $\psi'_d = \psi_d \circ \phi$. Let $t \in p(d)$ be the tuple resulting from ψ_d.

Assume that condition *2(a)* does not hold, i.e., there exists a node n' of p' such that $S_{at}(n', p') \not\subseteq S_{at}(\phi(n'), p)$. This means that n' has an attribute $a_{n'}$ that is not present in

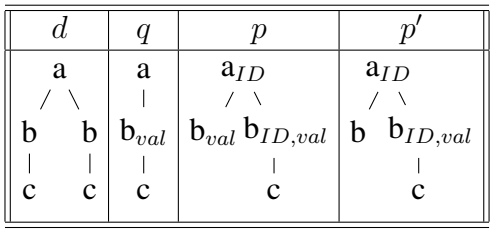


Figure 4.7: Tree pattern minimality.

$\phi(n') \in p$. Since $S_{at}(p) = S_{at}(p')$, there is another node n_p in p having an $a_{n'}$ attribute. Clearly $\phi(n') \neq n_p$ and, thus, n_p and n' are mapped to different nodes in d, giving p a different value for its $a_{n'}$ attribute than the value given to the one of p'. Hence, $t \notin p'(d)$.

Assume now that *2(b)* does not hold, that is, there are two p' nodes mapping to the same p node and having at least one attribute in common. As $S_{at}(p) = S_{at}(p')$, in a way similar to the one used above for condition *2(a)*, we can show that $t \notin p'(d)$. Finally, assume that condition *2(c)* does not hold, that is, a node n' in p' has a value predicate that is not present in $\phi(n')$ of p. We chose the d node corresponding to $\psi'_d(\phi(n'))$ such that its value does not satisfy this predicate. Thus, the result of $p'(d)$ based on ψ'_d is empty, and thus $t \notin p'(d)$. □

4.4.2 Tree Pattern Minimality

A node n of a tree pattern p is *redundant* if by removing n from p, the results of evaluating p over any document d remain the same.

Before further discussing about redundant nodes, we introduce the following notion:

Definition 4.4.4 (Structural endomorphism)**.** *Given a tree pattern p, a* structural endomorphism *is a structural embedding (as per Definition 4.4.3) from p to itself.*

It has been shown that in a tree pattern p with a single return node, a node $n \in p$ is redundant, *iff* there is a structural endomorphism on p that is not identity on n [AYCLS02].

It turns out that for tree patterns with multiple return node, such an endomorphism may exist even within a minimal pattern. For instance, consider the multiple return node patterns p and p' in Figure 4.7. Each of them has an isomorphism that is not the identity on their respective leftmost b nodes. However, the left b of p' is obviously redundant, and can thus be removed without affecting p semantics. In contrast, the leftmost b node in p is a return node and cannot be removed without affecting p semantics. In particular, p returns 4 tuples on the sample document d; if we removed the left b from p, we would get only 2 tuples.

We formalize the necessary and sufficient conditions for a node to be redundant in our tree pattern language, as follows. We say a node is *existential* if it is not a return node, and furthermore, none of its descendants are return nodes.

Proposition 4.4.2 (Redundancy test)**.** *A node n of a tree pattern p with multiple return nodes is* redundant*,* iff *(i) n is existential and (ii) there is a structural endomorphism on p that is not identity on n.*

Proof. We first prove if a node n is not existential, then it cannot be redundant. Either n is a return node, or it is non-return but has some return descendants. If n is a return node, clearly removing it will change p semantics, since even the pattern signature changes. Assume now that n is non-return, thus, it has a descendant n' that is a return node. As n' is return, it contributes to the results of $p(d)$ through its bindings on some d nodes. Removing n from p will change the path from the root of p to n' and, therefore, the bindings of n' on some documents. Thus, n is not redundant, because if removed, it changes the results of $p(d)$.

Having proved that the redundant nodes are always existential, the rest of the proof follows readily from the corresponding proof for tree patterns with single return nodes [AYCLS02]. □

A tree pattern is *minimal* if it does not include any redundant nodes.

Minimization algorithm A polynomial algorithm has been proposed for minimizing a tree pattern, running in $\mathcal{O}(n^4)$ in the worst case, where n the number of nodes [AYCLS02]. This algorithm can also be applied in our setting with the modification that if the leaf is non-existential (i.e., is a return node in this case), it is directly considered as non-redundant.

4.4.3 Tree Pattern Equivalence

Two tree patterns p, p' are *equivalent*, denoted $p \equiv p'$, if $p \sqsubseteq p'$ and $p' \sqsubseteq p$. We now introduce a more relaxed version of equivalence that is based only on the structural characteristics of the patterns.

Definition 4.4.5 (Structural equivalence)**.** *Two minimal tree patterns p and p' are* structurally equivalent*, denoted $p \overset{s}{\equiv} p'$,* iff *there exist $\phi : p \xrightarrow{s} p'$ and $\phi^{-1} : p' \xrightarrow{s} p$.*

Structural equivalence is necessary and sufficient for two tree patterns with single return nodes to be equivalent. In contrast, this is not the case for tree patterns with multiple return nodes, for which structural embeddings are not sufficient for containment, as shown in Section 4.4.1.

Theorem 4.4.3 (Tree pattern equivalence)**.** *Two minimal tree patterns p_1 and p_2 are* equivalent*, denoted $p_1 \equiv p_2$,* iff *there exists a structural embedding $\phi : p_1 \xrightarrow{s} p_2$ such that:*

1. *ϕ is bijective;*
2. *$\phi^{-1} : p_2 \xrightarrow{s} p_1$ is a structural embedding;*
3. *for each node n of p_1, $S_{at}(n, p_1) = S_{at}(\phi(n), p_2)$;*
4. *a node $n \in p_1$ has a value predicate $[=c]$,* iff *$\phi(n) \in p_2$ has the same predicate.*

Proof. ("If" direction) If the above conditions hold, it is easy to see that, according to Theorem 4.4.1, $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_1$, and thus, $p_1 \equiv p_2$.

("Only if" direction) Since $p_1 \equiv p_2$, we have $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_1$. Theorem Theorem 4.4.1 entails the presence of two structural embeddings $\phi_1 : p_1 \xrightarrow{s} p_2$ and $\phi_2 : p_2 \xrightarrow{s} p_1$.

Condition *1*: We need to show that ϕ_1 is both injective and surjective. We will first show that it is injective. Assume it is not. Let n_1, n_2 be nodes of p_1 and n' a node of p_2 such that $\phi_1(n_1) = \phi_1(n_2) = n'$. Moreover, let n_3 be a node of p_1 such that $\phi_2(n') = n_3$. Note also that the composition $\phi_{21} = \phi_2 \circ \phi_1$ is a structural endomorphism (see Definition 4.4.4) on p_1 and that $\phi_{21}(n_1) = \phi_{21}(n_2) = n_3$. We will show that both n_1 and n_2 are non-existential. Assume that n_1 is existential. If n_3 is different from n_1, Proposition 4.4.2 entails that n_1 is redundant (ϕ_{21} is an endomorphism different from the identity on n_1). However, p_1 is minimal by hypothesis, therefore, n_3 coincides with n_1. At the same time, n_2 has to be non-existential, otherwise it would be redundant (since $\phi_{21}(n_2) = n_1$). Moreover, given ϕ_1 and ϕ_2, Theorem 4.4.1 imposes the following relations:

$$S_{at}(n_1, p_1) \subseteq S_{at}(n', p_2) \tag{4.1}$$

$$S_{at}(n_2, p_1) \subseteq S_{at}(n', p_2) \tag{4.2}$$

$$S_{at}(n', p_2) \subseteq S_{at}(n_3, p_1) \xRightarrow{n_1 \equiv n_3} S_{at}(n', p_2) \subseteq S_{at}(n_1, p_1) \tag{4.3}$$

From (4.1) and (4.3) it follows that $S_{at}(n_1) = S_{at}(n')$, and since $S_{at}(n_1) = \emptyset$ (n_1 is existential and thus non-return), it holds $S_{at}(n') = \emptyset$. From (4.2) it follows that $S_{at}(n_2) = \emptyset$, i.e., n_2 is also non-return. But, as we have already shown, n_2 is non-existential and, thus, must have a return descendant, which is mapped through ϕ_1 to a return descendant n'' of n' in p_2. However, since n' is mapped to n_1 through ϕ_2, n'' has to be mapped to a return descendant of n_1. This is a contradiction, because n_1 is existential and cannot have return descendants. Therefore, n_1 is non-existential. We can similarly show that n_2 is not existential either.

We have thus shown that both n_1 and n_2 either are return nodes, or have return descendants. We now show that ϕ_1 is injective for the return nodes of p_1. In the proof of Theorem 4.4.1, we showed that if $p_1 \xrightarrow{s} p_2$, for every return node $n_2 \in p_2$ there is a node $n_1 \in p_1$ such that $n_2 = \phi_1(n_1)$. Since we also have $p_2 \xrightarrow{s} p_1$, it follows that p_1 and p_2 have the same number of return nodes, and, thus, ϕ_1 has to be injective for the return nodes (otherwise, some return nodes of p_2 would not be mapped by ϕ_1, which is not allowed).

Next, we show that ϕ_1 is injective also for the non-existential nodes that are non-return. To this regard, assume n_1 and n_2 are non-existential and non-return, mapped to the same n' node of p_2. We construct a document d such that there is an injective mapping ψ_1 from p_1 to d: for each node in p_1 we add a node in d with the same label; for each $/$-edge in p_1 we add an edge in d; for each $//$-edge (n_{11}, n_{12}) in p_1, we add a new node n_d in d with a fresh label that is not present neither in p_1 nor in p_2, and add the edges $(\psi_1(n_{11}), n_d)$ and $(n_d, \psi_1(n_{12}))$; each value predicate on a p_1 node is satisfied by the value of the corresponding d node. The embedding ψ_1 yields a tuple $t \in p_1(d)$ when p_1 is evaluated on d. One can embed p_2 into d through $\psi_1 \circ \phi_2$; this mapping embeds n' in a d node that is neither n_1 nor n_2, giving different values to the return descendants of n'. Thus, $t \notin p_2(d)$, which is a contradiction since $p_2 \sqsubseteq p_1$. Therefore, ϕ_1 is injective for all types of nodes.

Given that ϕ_1 is injective, we will show that it is also surjective, i.e., for every node $n_2 \in p_2$, there is a node $n_1 \in p_1$ such that $\phi_1(n_1) = n_2$. Assume this is not the case.

This, together with the fact that ϕ_1 is injective, entails that there is an additional node in p_2 that is not "covered" by ϕ_1. Then, one can construct a document d into which p_1 can be embedded but p_2 cannot. This leads to a contradiction, since $p_1 \sqsubseteq p_2$. Thus, ϕ_1 is bijective.

Condition *2*: Obviously, for every node n of p_2, n and $\phi^{-1}(n)$ have the same labels. Moreover, for every /-edge (n_1,n_2) in p_2, ($\phi^{-1}(n_1)$,$\phi^{-1}(n_2)$) is also a /-edge in p_1; otherwise, $\phi^{-1}(n_1)$ would not map through ϕ to n_1, and similarly $\phi^{-1}(n_2)$ would not map through ϕ into n_2. Finally, in a similar way, for every //-edge (n_1,n_2) in p_2, ($\phi^{-1}(n_1)$,$\phi^{-1}(n_2)$) is also a //-edge in p_1, and thus, there is a path from $\phi^{-1}(n_1)$ to $\phi^{-1}(n_2)$. Therefore, all conditions of Theorem 4.4.1 hold for ϕ^{-1} to be a structural embedding from p_2 to p_1.

Condition *3*: According to Theorem 4.4.1 and conditions *1* and *2*, for every node n of p_1, we have $S_{at}(n, p_1) \subseteq S_{at}(\phi(n), p_2) \subseteq S_{at}(\phi^{-1}(\phi(n)), p_1)$. Thus, $S_{at}(n, p_1) = S_{at}(\phi(n), p_2)$.

Condition *4*: If a node n of p_1 has a value predicate that $\phi(n)$ of p_2 does not have, it is easy to build a document in which p_2 can be embedded, but p_1 cannot, by virtue of the additional predicate. This is a contradiction since $p_2 \sqsubseteq p_1$. The same holds for predicates present in p_2 but not in p_1. □

4.4.4 Joined Tree Pattern Preliminaries

In this Section we discuss containment, minimality and equivalence of joined tree patterns.

Given a joined pattern jp and a set of documents $\mathcal{D}$, we denote by $jp(\mathcal{D})$ the result of evaluating jp on $\mathcal{D}$. A joined pattern jp' is *contained* into another joined pattern jp', denoted $jp \sqsubseteq jp'$, if $jp(\mathcal{D}) \subseteq jp'(\mathcal{D})$ for any document set $\mathcal{D}$. The joined patterns jp, jp' are *equivalent*, denoted $jp \equiv jp'$, if $jp(\mathcal{D}) \subseteq jp'(\mathcal{D})$ and $jp'(\mathcal{D}) \subseteq jp(\mathcal{D})$.

Definition 4.4.6 (Structural embedding)**.** *For a tree pattern t, let $\mathcal{N}_t$ denote the set of t nodes. Given two joined patterns jp and jp', a* structural embedding $\phi : jp' \xrightarrow{s} jp$ *is a function such that:*

1. *For every tree pattern p' of jp', the restriction of ϕ to $\mathcal{N}_{p'}$ is a structural embedding from p' to a tree pattern p of jp, i.e., $\phi|_{\mathcal{N}_{p'}} : p' \xrightarrow{s} p$;*
2. *If jp' has a value join*[1] *between n'_1 and n'_2, there also exists a value join between nodes $\phi(n'_1)$ and $\phi(n'_2)$.*

Let $\mathcal{T}_{jp} = \{p_i\}_m$ be the set of tree patterns of a joined pattern jp. We define the *signature of* jp as $S_{at}(jp) = \biguplus_{i=1}^{m} S_{at}(p_i)$. By $\uplus$ we denote the *bag union* of a set of multisets, i.e., the union that preserves the duplicate elements of the input multisets.

Theorem 4.4.4 (Joined pattern containment)**.** *Let jp and jp' be two joined patterns. $jp \sqsubseteq jp'$* iff*:*

1. Here we assume that the transitive closure of the value joins has been added to the pattern. For instance, if there are value joins between the nodes n_1, n_2 and n_2, n_3, we also add a value join between n_1, n_3.

1. $S_{at}(jp) = S_{at}(jp')$;
2. *there exists a structural embedding* $\phi : jp' \xrightarrow{s} jp$ *such that: if* $n_{p'}$ *is the root of a tree pattern* $p' \in jp'$*, then* $\phi(n_{p'}) = n_p$ *is the root of a tree pattern* $p \in jp$ *and* $p \sqsubseteq p'$.

Proof. ("If" direction) We have to show that if the above hypotheses hold, then $jp \sqsubseteq jp'$, i.e., if for a set of documents $\mathcal{D}$ there is a tuple $t \in jp(\mathcal{D})$, then $t \in jp'(\mathcal{D})$. Since $t \in jp(\mathcal{D})$, there is an embedding ψ_t from jp to $\mathcal{D}$. Pattern jp' can also be embedded to $\mathcal{D}$, through the composition $\psi'_t = \psi_t \circ \phi$. Indeed, according to hypothesis *2*, every tree pattern of jp' embeds into a document in $\mathcal{D}$. Moreover, since ψ_t respects the value joins of jp, ψ'_t respectively respect the value joins of jp'. At the same time, hypotheses *1* and *2* ensure that each return attribute of jp' can take values through ψ'_t from the same nodes of $\mathcal{D}$ that jp takes values. Thus, tuple $t \in jp'(\mathcal{D})$.

("Only if" direction) If $jp \sqsubseteq jp'$, condition *1* holds by definition. Assume that condition *2* does not hold. We will show that $jp \not\sqsubseteq jp'$, i.e., for a document set $\mathcal{D}$ there exists a tuple $t \in jp(\mathcal{D})$ such that $t \notin jp'(\mathcal{D})$. There exist two tree patterns p of jp and p' of jp' such that $p \not\sqsubseteq p'$. Based on this, we can construct a document set such that a tuple $t \in jp(\mathcal{D})$ contains in the positions of the attributes that correspond to p some values that cannot be taken by $jp'(\mathcal{D})$ (those values that make $p \not\sqsubseteq p'$). Thus, $t \notin jp(\mathcal{D})$, which is a contradiction. □

Proposition 4.4.5 (Redundancy check)**.** *A node* n *of a joined pattern* jp *is redundant,* iff *the node corresponding to* n *in the extended pattern* jp^x *(recall Definition 4.3.1) is redundant in* jp^x.

Proof. The proof follows from Proposition 4.4.2 and the following remark. Let p be the jp tree pattern to which n belongs. Observe that in order to be redundant in jp, n must also be redundant in p. Now let us consider the possibility that n, or one of its descendants, participates in a value join in jp; if this is the case, n is not redundant for jp, even if it is redundant for p. In more detail:

- If n participates in a value join and we remove n, then the value join is no longer ensured.
- If a descendant n' of n participates in a value join and by removing n we obtain the new joined pattern jp', the path to n' in jp' is less restrictive (since node n was removed) than the path above the corresponding node in jp. Thus a document can be constructed in which jp' can be embedded, but the more restrictive jp cannot. Thus, jp and jp' do not yield the same results on every document and n is not redundant.

The above two conditions are ensured by checking if e^x is redundant for the tree pattern t^x of jp^x to which it belongs. In fact, by using the extended version of jp, we turn all nodes that them or one of their descendants participate in a value join to non-existential, making sure that they will not be characterized as redundant. □

A joined pattern is *minimal* if it does not contain any redundant nodes. Observe that minimality concerns only nodes and not value join edges, whose transitive closure is assumed to be present in the pattern.

Minimization algorithm To minimize a joined pattern jp, we build its extended version jp^x and run the minimization algorithm for the tree patterns (described in Section 4.4.2) for each tree pattern of jp^x. Let jp^x_{min} be the resulting joined pattern. Then, the joined pattern jp_{min} obtained from jp^x_{min} by removing the additional attributes introduced in jp^x is, according to Proposition 4.4.5, the minimized version of jp.

Theorem 4.4.6 (Joined pattern equivalence)**.** *Two minimal joined patterns jp_1, jp_2 are* equivalent *($jp_1 \equiv jp_2$),* iff *there exists a structural embedding $\phi : jp_1 \xrightarrow{s} jp_2$ such that:*

1. *ϕ is an isomorphism;*
2. *$\phi^{-1} jp_2 \xrightarrow{s} jp_1$ is a structural embedding;*
3. *if n_{p_1} is the root of a tree pattern $p_1 \in jp_1$, then $\phi(n_{p_1}) = n_{p_2}$ is the root of a tree pattern $p_2 \in jp_2$ and $p_1 \equiv p_2$.*

Sketch. ("If" direction) If the above hypotheses hold, then according to Theorem 4.4.4, it can be easily shown that $jp_1 \sqsubseteq jp_2$ and $jp_2 \sqsubseteq jp_1$. Thus, $jp_1 \equiv jp_2$.
("Only if" direction) Condition *1*: We prove condition *1* in the same way as we proved condition *1* of Theorem 4.4.3.
Condition *2*: According to the proof of Theorem 4.4.3, for every tree pattern p_1 of jp_1, mapped to a tree pattern p_2 of jp_2 through ϕ, the inverse of the restriction of ϕ to the node set $\mathcal{N}_{p_1}$ of p_1, denoted $\phi|_{\mathcal{N}_{p_1}}{}^{-1}$, defines a structural embedding from p_2 to p_1. Moreover, there is a value join between nodes n_{11} and n_{12} of jp_1, *iff* there is a value join between $\phi(n_{11})$ and $\phi(n_{12})$ in jp_2. If this is not the case, then the set of value joins of the one joined pattern is more restrictive than the other's one, and a document set for which the two joined patterns do not return the same results can be built. From this and Definition 4.4.6 follows that $\phi^{-1} : jp_2 \xrightarrow{s} jp_1$.
Condition *3*: We have shown that ϕ is an isomorphism and defines a two-way structural embedding between jp_1 and jp_2. Let p_1 be a tree pattern of jp_1 which ϕ_1 maps to tree pattern p_2 of jp_2. The restriction $\phi|_{\mathcal{N}_{p_1}}$, where $\mathcal{N}_{p_1}$ is the node set of p_1, is also an isomorphism defining a two-way structural embedding between p_1 and p_2. Moreover, for jp_1 and jp_2 to give the same results on every document set, every node n_1 of p_1 should have the same signature as $\phi(n_1)$ in p_2, and also have the same value predicates. Then, according to Theorem 4.4.3, $p_1 \equiv p_2$. □

4.4.5 DAG Pattern Preliminaries

Although our views and queries can only be expressed as (joined) tree patterns, the intermediate products of our rewriting algorithm may temporarily take the form of DAG patterns. In this Section we discuss containment of such patterns, as well as how a DAG can be transformed to a tree.

DAG patterns and semantics A DAG pattern dp is a directed acyclic graph, whose nodes carry labels from $\mathcal{L}$ and are annotated with zero or more among the attributes ID, val and $cont$. A dp node may also be annotated with a value equality predicate. Moreover, each DAG pattern has a unique *root* node. Unlike tree patterns, for each node n of a DAG pattern dp, there may be more than one incoming paths from the root of dp to n.

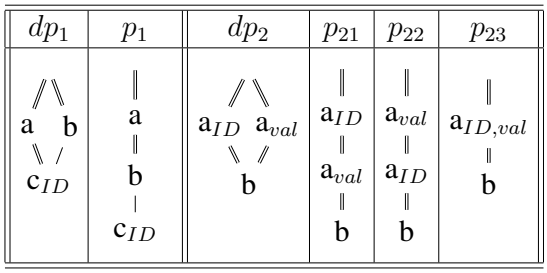


Figure 4.8: DAGs and interleavings.

The semantics of a DAG pattern dp is defined in the same way as the semantics of a tree pattern, with the difference that when there is an embedding from dp to a document tree t, all incoming paths to a dp node n should be satisfied by the embedding.

DAG pattern containment Definitions 4.4.1, 4.4.2, 4.4.3 of tree pattern signature, containment and structural embedding, respectively, carry directly to DAG patterns. Moreover, Theorem 4.4.1 can also serve as the basis for deciding containment between DAG patterns dp_1, dp_2 (denoted $dp_1 \sqsubseteq dp_2$).

Interleavings We now introduce the notion of interleavings that will be then used to express a DAG pattern with multiple return nodes as a union of tree patterns. The interleavings for tree patterns with single return nodes were introduced in [BFK05] and were also used in [CDO08]. In those cases, only the single return node of the DAG pattern was allowed to have multiple incoming paths. Informally, an interleaving is a tree pattern that respects the labels and the relationships of all paths of the respective DAG. For instance, in Figure 4.8, dp_1 is a DAG pattern and p_1 is one of its interleavings (in fact, in this case it is its only interleaving).

In what follows, we describe how an interleaving can be built in our case, where every node is allowed to have multiple incoming paths. We term such nodes *multi-incoming* nodes.

Let dp be a DAG pattern and n be a multi-incoming node in dp, none of whose ancestors are multi-incoming. We replace the set of paths $\{p_{n_i}\}$ from the root of dp to n with a new single path p'_n such that:

1. there exists a surjective function ϕ_i, mapping each node of $\{p_{n_i}\}$ to a p'_n node (ϕ_i does not need to be injective);
2. for each node n of $\{p_{n_i}\}$, n and $\phi_i(n)$ have the same label;
3. for each $/$-edge (n_1, n_2) in $\{p_{n_i}\}$, $(\phi_i(n_1), \phi_i(n_2))$ is a $/$-edge in p'_n, and for each $//$-edge (n_1, n_2) in $\{p_{n_i}\}$, there is a path from $\phi_i(n_1)$ to $\phi_i(n_2)$ in p'_n;
4. when one or more nodes of $\{p_{n_i}\}$ are mapped to the same p'_n node n_p, n_p is annotated with the bag union of the attributes of these nodes [2];

2. If two such nodes have the same attribute, this attribute will appear twice in n_p with the obvious semantics of outputting it twice in the resulting tuples as well. This way p'_n has the same signature with the union of signatures of the paths it replaces.

5. the outgoing edges of each node n of $\{p_{n_i}\}$ that do not belong to $\{p_{n_i}\}$, are attached to $\phi_i(n)$ in p'_n;
6. if n has a value predicate, $\phi_i(n)$ has the same predicate. Moreover, if two dp nodes are mapped to the same p node and they both have a value predicate, they should have the same one.

We repeat the above process, each time replacing the incoming paths of a multi-incoming node with a single path, until there are no more multi-incoming nodes. The resulting tree pattern is an inverleaving p_i for dp.

For instance, in Figure 4.8, dp_2 is a DAG pattern with multiple return nodes, and p_{21}, p_{22}, p_{23} are interleavings of dp_2.

We now use the interleavings to express a DAG as a union of tree patterns. This result follows from a similar result stated in [BFK05, CDO08].

Lemma 4.4.7. *A DAG pattern is equivalent to the union of its interleavings.*

Similar to a result from [CDO08], we also have that:

Proposition 4.4.8. *If a tree pattern p is equivalent to a DAG pattern dp, then p is equivalent to one of dp's interleavings.*

Proof. (Sketch) According to Lemma 4.4.7, dp is equivalent to the union of its interleavings. For dp to be equivalent to a tree pattern p, a member of the union of its interleavings should contain all the rest and be equivalent to p. □

Notice that in Figure 4.8, dp_2 cannot be transformed to a tree pattern, because none of its interleavings contains the rest.

It has been previously shown [BFK05] that there may be exponentially many interleavings for an XPath 1.0 tree pattern, and thus for our tree pattern language.

We define the transformation **DAG2Tree**(d) that takes as input a DAG pattern d and transforms it into an *equivalent* tree pattern p, if one such pattern exists, or fails if none can be found. **DAG2Tree** starts by expressing d as the union of its interleavings (Lemma 4.4.7), and then tries to find an interleaving that contains all the others (Proposition 4.4.8). If one is found, **DAG2Tree** returns it.

4.5 Tree Pattern Query Rewriting Overview

This Section outlines our algorithm for finding view-based query rewritings (see Section 4.3.3 for formal problem definition), for the case when the query and views correspond to single tree patterns. From the XQuery syntax viewpoint, this restriction amounts to replacing rules 1 and 4 in the grammar of Figure 4.2 with:

$1'$	q :=	for $absVar$ (, $relVar$)* (where $pred$ (and $pred$)*)? return ret
$4'$	$pred$:=	string(x_i) =c

Algorithm 3: Tree Pattern Rewriting (**TPR**)

Input : View set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, query q
Output: All minimal algebraic rewritings e of q using $\mathcal{V}$

$S_1 \leftarrow \emptyset$; $S_{crt} \leftarrow \emptyset$; $S_{spo} \leftarrow \emptyset$
//Filter views and add the useful (p, e, ϕ) to S_1
foreach $v \in \mathcal{V}$ **do**
  foreach *embedding* $\phi : v \xrightarrow{s} q$ **do**
    if $\mathbf{ViewFilter}(q, v, \phi)$ **then**
      //Check if v is directly a solution
      if $v \equiv q$ **then** output solution $scan(v)$
      else add $(v, scan(v), \phi)$ to S_1
//Attempt single view rewritings
foreach $(p_1, e_1, \phi_1) \in S_1$ **do**
  $S_{spo} \leftarrow \mathbf{TPTransform}(q, p_1, e_1, \phi_1)$
  foreach $(p_s, e_s, \phi_s) \in S_{spo}$ **do**
    if $p_s \equiv q$ **then** output solution e_s
    else add (p_s, e_s, ϕ_s) to S_{crt}
$S_1 \leftarrow S_{crt}$
//Join partial rewritings
while *new* (p, e, ϕ) *triples are added to* S_{crt} **do**
  $((p_1, e_1, \phi_1), (p_2, e_2, \phi_2)) \leftarrow \mathbf{PickPair}(q, S_1, S_{crt})$
  $(p, e, \phi) \leftarrow \mathbf{TPJoin}(q, (p_1, e_1, \phi_1), (p_2, e_2, \phi_2))$
  if $(p, e, \phi) \neq null$ **then**
    $S_{spo} \leftarrow \mathbf{TPTransform}(q, p, e, \phi)$
    foreach $(p_s, e_s, \phi_s) \in S_{spo}$ **do**
      if $p_s \equiv q$ **then** output solution e_s
      else add (p_s, e_s, ϕ_s) to S_{crt}

We consider the view-based query rewriting for the full language in Section 4.8.

In Section 4.5.1 we provide an overview of the tree pattern rewriting algorithm, whereas in Section 4.5.2 we identify some interesting classes of rewritings, on which we focus our search.

4.5.1 TPR Algorithm Overview

Our tree pattern rewriting algorithm, called **TPR**, takes as input a tree pattern query q and a set of tree pattern views $\mathcal{V}$, and outputs all minimal algebraic rewriting of q based exclusively on $\mathcal{V}$. It is outlined in Algorithm 4 and consists in three stages: the *view filtering*, *single view rewriting*, and *joining partial rewritings* stages. Here we provide a brief overview of the whole algorithm; individual steps will be detailed in Section 4.7, along with completeness and termination results, as well as optimizations.

The algorithm starts by identifying all possible ways in which a view could be used to answer query q, that is, all possible structural embeddings of each view to q (line 3). Then, the *view filtering* (line 4) step discards views which *cannot* appear in a rewriting, based on some pruning criteria, as we will detail in Section 4.7. At this point, if a view is directly equivalent to the query, it is output as a solution.

We then build the S_1 set, containing, for each view v defined by the tree pattern p, and embedding ϕ from v into q, triples of the form *(view tree pattern p, algebraic expression e, embedding ϕ)*, where e is a scan over v. Clearly, the pattern p and the expression e are equivalent, that is, they return the same data regardless of the content of the XML database. It turns out that throughout its execution, algorithm **TPR** only needs to manipulate (pattern, expression, embedding) triples such that the pattern is a tree pattern and is equivalent to the algebraic expression; this will be explained in Section 4.5.2.

During the *single view rewriting* stage (lines 7-11), the algorithm applies various transformations on each (p, e, ϕ) triple of S_1 (**TPTransform** function). Each transformation (i) adds one or several algebraic operators on top of e, while simultaneously (ii) altering p to keep it equivalent to the modified algebraic expression and (iii) modifying ϕ accordingly, to embed the modified p into the query. Sample transformations aim at adding, for example, a value selection on a view, or a selection on node IDs to transform a // edge into a / edge etc. Modifying (pattern, expression, embedding) triples, all the while keeping them in sync is a pretty complex process, which we discuss in Section 4.6.

This stage, thus, produces (pattern, expression, embedding) triples, such that the expression is based on a single view. If the pattern is equivalent to the query, the algebraic expression is an equivalent rewriting of the query using only that view. Otherwise, the triple produced by **TPTransform** is considered a *partial rewriting* based on which to continue the search, as we explain below.

During the *joining partial rewritings* stage (lines 13-20), partial rewritings are combined using equality joins on node IDs (and, if IDs are structural, by structural joins as well), by applying the **TPJoin** operation. To further refine the result of a join by adding necessary constraints from the query, we may call the **TPTransform** function again during this stage. The order in which the partial rewritings are combined is encapsulated within the function **PickPair**; we will discuss concrete orders in Section 4.7.

Importantly, during this stage, we build a join of two partial rewritings, only if the join result is closer to an equivalent query rewriting than each of the inputs to the join; in other words, by design we only build minimal algebraic expressions. This has two advantages: (i) reducing the algorithm running time since we only enumerate minimal join expressions; (ii) producing minimal rewritings only, which, under the cost hypothesis we consider (Section 4.3.3), lead to lower evaluation costs.

4.5.2 Left-Deep Query Tree Organized Rewritings (LDQT)

This Section makes two crucial observations on the space of partial or equivalent rewritings. Each observation leads to identifying a subset of all possible rewritings, and shows that it suffices to search within this class, while still preserving completeness.

We first observe that algebraic expressions (and in particular, rewritings of a query

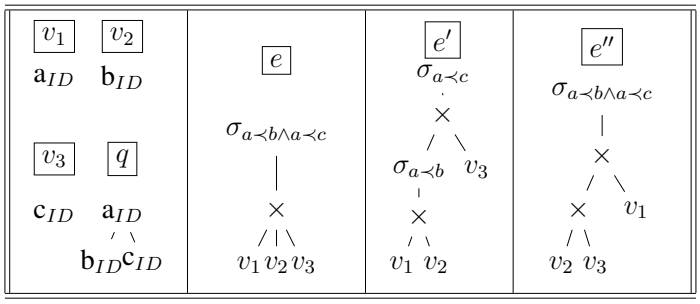


Figure 4.9: Sample views, query, non-left-deep rewriting, and two corresponding left-deep rewritings.

based on the same set of views) may differ in the details of their algebraic syntax, yet represent fundamentally the same (equivalent) rewriting. For instance, if $e = \sigma_{cond}(v_1 \bowtie (v_2 \bowtie v_3))$ is a rewriting, where $cond$ is a predicate on the view v_1, then so is $e' = ((\sigma_{cond}(v_1) \bowtie v_2) \bowtie v_3)$. Our search for rewritings should not spend time enumerating candidate rewritings that can be obtained from one another by pushing σ and π operators, exploiting the transitivity of the $=$, $\prec$ and $\prec\!\!\prec$ comparison operators, re-ordering joins etc. Instead, such transformations should be left to the subsequent optimization stage, and rewriting should focus on finding fundamentally different alternatives.

Left-deep rewritings formalize this intuition:

Definition 4.5.1 (Left-deep rewriting)**.** *A rewriting e of the query q is* left-deep iff*: (i) all $\times$ operators in e are binary, and their right-hand children contain no $\times$ operator (e is a left-deep binary tree); (ii) all σ, π and δ are pushed as low as possible in e; (iii) all the nav operators are applied below any $\times$ operator.*

From a rewriting e, one can obtain by algebraic transformations several left-deep rewritings, as illustrated in Figure 4.9, where for readability, we write $a \prec b$ instead of $a.ID \prec b.ID$ (and similarly for other predicates). Note also that here and in the sequel, we may omit drawing the edge above a top pattern node, whenever the discussion does not require it. In Figure 4.9, e' and e'' are left-deep rewritings obtained from e. We call the set of left-deep rewritings obtainable from a rewriting e via algebraic transformations, the *corresponding* left-deep rewritings of e.

Our second observation exploits the inner connection between tree patterns and algebraic operators. We first introduce:

Definition 4.5.2 (LDT rewriting)**.** *A left-deep rewriting is tree-organized (LDT rewriting, in short),* iff *each sub-plan p of e, such that p is a child of a $\times$ operator, is equivalent to some tree pattern t_p.*

In Figure 4.9, e' is an LDT rewriting, since the leaf operators corresponding to scans of the views v_1, v_2 and v_3 are equivalent to the respective view tree patterns. Moreover, the plan $\sigma_{a \prec b}(v_1 \times v_2)$ is equivalent to the tree pattern //a_{ID}/b_{ID}. In contrast, e'' in Figure 4.9 is not an LDT rewriting: its sub-plan $v_1 \times v_2$ is a child of a $\times$ operator, and is not equivalent to any tree pattern, since it combines data from unrelated nodes.

We now state an important property:

Proposition 4.5.1. *Let e be any rewriting of q using $\mathcal{V}$. Then, there exists an LDT rewriting e' corresponding to e.*

Proof. We start by proving the existence of the LDT rewriting e'.

Let $e_1, e_2, \ldots, e_l$ be the set of left-deep rewritings corresponding to e, and assume that in all e_i, $1 \leq i \leq k$, there are some $\times$ children, which are not equivalent to any pattern. For a given e_i, let p_i be a *lowermost* such sub-tree, i.e., such that all descendant of p_i, are equivalent to some tree pattern (p_i is guaranteed to exist since the leaves of e_i are view scans.)

If p_i is not equivalent to any tree pattern, this means that p_i's output tuples have information coming from at least two *unrelated* view nodes, i.e., having no common ancestor. (An example is the left-hand child of the $\times$ operator in the e'' rewriting of Figure 4.9: it returns IDs of unrelated bs and cs.) Without loss of generality, let us assume there are exactly two such unrelated nodes, x and y. On the path from p_i to the root of e_i, let p_j be the lowermost operator (either a child of $\times$, or the root of e_i itself) such that p_j is equivalent to a tree pattern. (Clearly, p_j exists, because e_i is equivalent to q.) For p_j to transform its non tree-pattern-equivalent input to some tree-pattern-equivalent output, it has to *relate* data from the unrelated nodes x and y, by ensuring they have a common ancestor, say z. Thus, p_j must be a selection, which applies at least two predicates of the form $z \prec x$ (or $z \prec\!\!\!\prec x$) and $z \prec y$ (or $z \prec\!\!\!\prec y$), such that z is a view node which does not contribute to p_x (if z contributed to p_x, then these predicates would have been pushed down by the definition of left-deep rewritings, and p_x would then be equivalent to a tree pattern). In Figure 4.9, p_j is the σ operator in e''.

We now identify the following operator nodes in e_i:

p_z the lowermost $\times$ child, descendant of p_j, which already has the attributes in p_j's input, coming from the view node z. (In Figure 4.9, p_z is v_1.)

p_x the lowermost $\times$ child, descendant of p_i, which has all the attributes in p_i's input(s), coming from the view node x. (In Figure 4.9, p_x is v_2.) By choice of p_i, p_x is equivalent to a tree pattern.

p_y the lowermost $\times$ child, descendant of p_i, which has all the attributes in p_i's input(s), coming from the view node y. (In Figure 4.9, p_y is v_3.) Similarly to p_x, p_y is also equivalent to a tree pattern.

We build an expression e' by copying e_i and in this copy, swapping the subtrees p_x, p_y and p_z of e_i. More specifically:

1. we add a $\times p_z$ branch immediately above p_x, and on top of the $\times$ we push all p_j operations (σ, π, de) which can be evaluated here; the result will be equivalent to a tree pattern, since p_x was equivalent to one, and we have pushed all the operators from p_i up to p_j which related x to z (the nodes n_x and n_z are related);
2. on top of this, we copy all operators from p_x to p_i, until (and including) the $\times$ connecting it with p_y;
3. on top of this $\times$, we push all operators from p_i which can apply now; the result will be equivalent to a tree pattern (by a similar reasoning);

4. we remove the p_z subtree from its initial place under p_j since it has been pushed down.

Applying this procedure to e'' in Figure 4.9 yields the rewriting e', in which each $\times$ child is equivalent to a tree pattern.

Coming back to the proof, it is easy to see that e' is equivalent to e_i, since it has been obtained by re-ordering the branches of e_i; and, e' is also left-deep. Thus, we have obtained a left-deep rewriting corresponding to e, in which the lowermost node non-equivalent to a tree pattern is higher than the one in e_i.

By repeating the above procedure, we can lift the p_i node arbitrarily high in e'. Given that the root of e' is equivalent to the (query) tree pattern, it follows that there is always a left-deep rewriting corresponding to e, such that all its subtrees are equivalent to some tree pattern. □

For example, in Figure 4.9, the rewriting e' is LDT and corresponds to e. Observe that e' does not need to be unique: swapping v_2 with v_3 and b with c in the predicates of e' in Figure 4.9 yields another LDT rewriting e''', in which all $\times$ children operators are equivalent to some tree pattern.

An important subset of LDT rewritings consists of:

Definition 4.5.3 (LDQT rewriting)**.** *An LDT rewriting e is query-tree-organized (LDQT in short)* iff *for each sub-plan p of e, such that p is a child of a $\times$ operator, and p is equivalent to the tree pattern t_p, t_p can be embedded into q.*

For example, the LDT rewriting e' in Figure 4.9 is an LDQT rewriting, since the tree pattern equivalent to $\sigma_{a \prec b}(v_1 \times v_2)$ is the left branch of q. In this example, a sample LDT rewriting which is not LDQT, would be $\sigma_{b \prec a}(v_1 \times v_2)$, equivalent to the tree pattern //b_{ID}/a_{ID}, which cannot be embedded in q.

Proposition 4.5.2. *Let e be a rewriting of q. There exists an LDQT rewriting e' corresponding to e.*

Proof. Let p be a $\times$ child in e' such that its equivalent tree pattern t cannot be embedded in q. Then, t (and its equivalent p) enforces some condition that is not present in the query (such as a node which does not appear in the query, or two nodes which do appear in the query but in a structural relationship contradicting the one in the query). Then, this condition will still hold on the output of e', since no algebraic operator between p and e' can "lift" it; therefore, the rewriting e' is not equivalent to the query (contradiction). □

Observe that several LDQTs may correspond to a given rewriting, e.g., in Figure 4.9, e' and e''' mentioned above are LDQTs corresponding to e.

Proposition 4.5.2 is of crucial importance, as it allows us to *build and use only LDQT rewritings* during the rewriting. Any non-LDQT rewriting e can be derived by the optimizer from its corresponding LDQT rewriting e', by reversing the algebraic transformations which compute e' from e.

p_1	p_2	$p_1' \leftarrow ValPredAd(p_1, p_2)$	p_3	p_4	$p_3' \leftarrow nav_{a.cont,p_4}(p_3)$
a / \ b_{val} c	a / \ b c [=5] [=7]	a / \ b_{val} c [=5]	a_{cont} | b	a / | \ b c_{ID} d_{val}	a_{cont} / |\ b c d_{val}

Figure 4.10: Value predicate adaptation and navigation.

4.6 Algebraic Transformations on Tree Patterns and DAGs

In this Section, we consider the problem at the core of the **TPTransform** function used by the rewriting algorithm: given a (tree or DAG) pattern p and a *target* tree pattern p', how can one *transform* p *into* p' by means of algebraic operations? More precisely, we are given the pattern p, and an algebraic expression e equivalent to p, and we are interested in ways of applying algebraic operations on e (while also modifying p to keep it equivalent to the modified expression) until they are both equivalent to p' and thus, the expression is a way to compute p'. To ease reading, our discussion is organized into subsections, each corresponding to a type of transformation one could apply on p.

4.6.1 Value Predicate Adaptation (ValPredAd)

We first discuss how value equality predicates can be algebraically added to a tree pattern. Let p_1, p_2 be two tree patterns, such that there is a structural embedding (see Definition 4.4.3) $\phi : p_1 \xrightarrow{s} p_2$, and let e_1 be the corresponding algebraic expression of p_1. We define the transformation **ValPredAd**(p_1,p_2), which computes modified versions of p_1 and e_1 as follows. **ValPredAd** visits each node $n_1 \in p_1$, and for each value predicate of the form $[= c]$ attached to $\phi(n_1) \in p_2$ and not attached to n_1:

- if n_1 is val-return, the predicate $[=c]$ is added to n_1 and e_1 is substituted by $\sigma_{n_1.val=c}(e_1)$, i.e., a selection is added on top of e_1;
- if n_1 is not val-return, both p_1 and e_1 are left intact and the function continues to the next node.

ValPredAd returns: (i) the pattern p_1' obtained by modifying p_1; (ii) the (equivalent) algebraic expression e_1' obtained from e_1; (iii) the embedding $\phi' : p_1' \xrightarrow{s} p_2$, which is the same as ϕ (p_1 and p_1' have the same structure).

As an example consider patterns p_1 and p_2 of Figure 4.10, with p_2 having two value predicates not present in p_1. In this case, **ValPredAd** adds a value predicate on p_1's b node, but not on its c node, since the latter is not val-return.

4.6.2 Navigation (Nav)

Since a tree pattern (view) node annotated with $cont$ denotes the fact that complete XML subtrees matching that view node are stored in the view, we may extend the view to add more (possible return) nodes. This is done through the nav operator (see Sec-

tion 4.3.2) and has been first introduced as a technique in one-view rewritings of tree patterns with single return nodes [XO05].

Let p_1, p_2 be two tree patterns, such that there exists $\phi : p_1 \xrightarrow{s} p_2$, and let e_1 be the algebraic expression corresponding to p_1. We define the **Nav**(p_1,p_2) transformation, which returns modified versions of p_1 and e_1 as follows. **Nav** visits each $cont$-return node $n_1 \in p_1$ and applies a nav operator on e_1, requiring that we navigate inside the $n_1.cont$ attribute to retrieve matches for a pattern p', defined as follows. Pattern p' is a copy of p_2's sub-pattern rooted in $\phi(n_1)$, from which we remove all ID stored attributes (while preserving all val and $cont$ attributes). IDs are treated differently since in our model, we assume node IDs are not part of $cont$ values, the latter corresponding to serialized XML subtrees. Each $nav_{n_1.cont,p'}$ operator thus introduced has the effect of adding to p_1 all the nodes found under $\phi(n_1)$, along with their attributes (modulo the ID issue above).

Nav returns: (i) a new tree pattern p'_1 enriched with all the nodes introduced by the sequence of nav operators; (ii) an algebraic expression e'_1 of the form $nav(nav(\ldots(e_1)\ldots))$, equivalent to p'_1; (iii) an embedding $\phi' : p'_1 \xrightarrow{s} p_2$ that is the same as ϕ for the nodes of p'_1 that also appeared in p_1, and that maps 1-1 each of the additional nodes of p'_1 to the corresponding nodes of p_2.

Nav is illustrated by an example in Figure 4.10. Applying **Nav** on p_3 and p_4 leads to navigate within p_3's $a.cont$ attribute, leading to the pattern p'_3.

4.6.3 Attribute Elimination (AttElim)

We now consider the problem of eliminating some of the attributes of a tree pattern p in order to obtain a pattern p' that is structurally equivalent to p, ($p \overset{s}{\equiv} p'$, see Definition 4.4.5), but p' contains only a subset of p's attributes. Note that all results of this Section carry over to the more general case of DAG patterns as well.

Clearly, moving from p to p' leads to *fewer attributes* in the relations corresponding to the pattern (or view) content. It *may* also lead to *fewer tuples*. To see why, consider the patterns $p = //a_{ID}//b_{val}$, $p' = //a_{ID}[//b]$, and the document <a><b1/><b2/>, where b1 denotes the first b element and b2 denotes the second. In this case, p and e compute the tuple set {(a.ID, b1.val), (a.ID, b2.val)} while p' returns {(a.ID)}, i.e., p' indeed returns less tuples than p. This happens because in p' the b node has become existential and the number of results is determined only by the return node a.

It is important to note that for any document d, since p and p' are structurally equivalent, they can be embedded in exactly the same ways to d. More formally, for any embedding $\phi : p \to d$, there exists an embedding $\phi' : p' \to d$, such that for any $n \in p$ and the corresponding node $n' \in p'$, we have $\phi(n) = \phi(n')$. The opposite also holds (for any embedding $\phi' : p \to d$, there is an embedding ϕ such that for any $n \in p$ and the corresponding node $n' \in p'$, $\phi(n) = \phi'(n')$).

This tight connection between the ways p and p' can be embedded in the database, together with the fact that any return node of p' is also a return node in p, guarantees that a tuple in $p'(d)$ can only take values that also appear together in some p tuple. The following Lemma formalizes the above remarks; the proof is quite straightforward and thus we omit it.

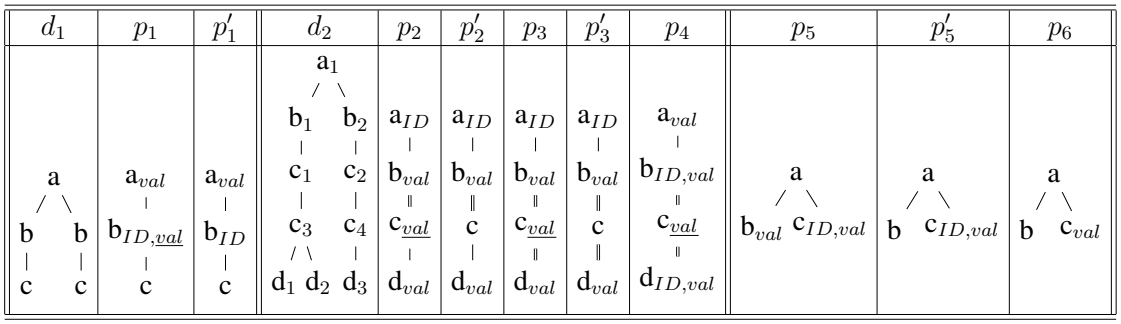


Figure 4.11: Attribute elimination.

Lemma 4.6.1 (Tuples resulting from attribute elimination)**.** *Let p, p' be two patterns such that $p \overset{s}{\equiv} p'$ and for any node $n \in p$ and corresponding node $n' \in p'$, the return attributes of n' are a subset of the return attributes of n. Then, for any document d:*

1. *for any tuple $t' \in p'(d)$, there exists a tuple $t \in p(d)$ such that t restricted to t''s attributes is exactly t', in other words: $\pi_{p'}(t) = t'$, where $\pi_{p'}$ designates a projection retaining exactly the return attributes of p'*
2. *for each tuple $t' \in p'(d)$, such that t' appears $n_{t'}$ times in $p'(d)$, let T be the set of tuples $\{t \in p(d) \mid \pi_{p'}(t) = t'\}$ and for each $t \in T$, let n_t be the number of times that t appears in $p(d)$. Then, we have:*

$$n_{t'} \leq \sum_{t \in T} n_t$$

In the example at the beginning of this Section, taking $t' =$(a.ID), we obtain $T = \{$(a.ID, b1.val), (a.ID, b2.val)$\}$. We have $n_t = 1$ and $\sum_{t \in T} n_t = 2$.

Let e be the algebraic expression corresponding to p. Lemma 4.6.1 entails that in order to adjust e into an expression e' corresponding to p', one must (i) *necessarily remove some columns* from e, and (ii) *possibly reduce the multiplicity of some tuples* from e.

In our algebraic toolbox (Section 4.3.2.3), *projection* is the obvious tool for removing columns, whereas *duplicate elimination* is the tool for adjusting the tuple multiplicity. In the relational setting, under *set semantics*, a projection suffices to eliminate an attribute. Under *bag semantics*, a duplicate elimination is also needed on top of the projection. As explained in Section 4.3, the semantics of our views carries the W3C's intended XPath and XQuery semantics into a tuple-based setting, where it is defined by the number of bindings (embeddings) of the return nodes of the tree pattern on a given document. Under this semantics, although projection is used to remove columns, the situations when a duplicate elimination is needed are not obvious. Moreover, when duplicate elimination *is* necessary, it may not be sufficient in order to adjust the multiplicity of the results, all the while attaining the exact semantics of the target pattern p'.

A simple example illustrates projection. In Figure 4.11, to transform p_2 into p'_2, one only needs to project out the $c.val$ attribute. However, some other cases raise more difficulties. Consider pattern p_3 in Figure 4.11, for which $p_3(d_2)$ returns the following six

tuples [3]:

$a.ID$	$b.val$	$c.val$	$d.val$
a_1.ID	b_1.val	c_1.val	d_1.val
a_1.ID	b_1.val	c_1.val	d_2.val
a_1.ID	b_1.val	c_3.val	d_1.val
a_1.ID	b_2.val	c_3.val	d_2.val
a_1.ID	b_2.val	c_2.val	d_3.val
a_1.ID	b_2.val	c_4.val	d_3.val

Assume that our goal is to eliminate from p_3, the val attribute of the c node (underlined in the Figure), in order to obtain the pattern p'_3 in the Figure. Thus, assuming e is an algebraic expression corresponding to p_3, we are searching for an algebraic expression of the form $\alpha(e_3)$, which must be equivalent to p'_3. A first option is $e' = \pi(e_3)$, where the projection removes the undesired $c_{\underline{val}}$ attribute. However, e' is not equivalent to p'_3: while e' returns six tuples (a simple projection does not remove duplicates), $p'_3(d_2)$ returns only the following three tuples:

$a.ID$	$b.val$	$d.val$
a_1.ID	b_1.val	d_1.val
a_1.ID	b_1.val	d_2.val
a_1.ID	b_2.val	d_3.val

A second option then is to take $e'' = \delta(\pi(e'))$, where δ is a duplicate elimination operator. Nevertheless, depending on the actual values carried by the nodes of d_2 (values are not shown in the Figure), e'' may return anywhere from one to three tuples. For instance, if $b_1.val = 2$, $b_2.val = 4$, $d_1.val = d_2.val = 5$ and $d_3.val = 7$, then e'' returns two tuples. On the contrary, if $b_1.val = b_2.val$ and $d_1.val = d_2.val = d_3.val$, e' outputs only one tuple.

The last example has shown that adapting the algebraic expression e corresponding to a pattern p, in order to get an expression equivalent to a given different pattern p', is quite involved. It turns out that in some cases, it is impossible, that is: starting from a tree pattern p and equivalent algebraic expression e, there is no algebraic expression $e' = \alpha(e)$ such that e' be equivalent to a pattern p' obtained from p by removing some of p's return attributes.

In the sequel, we address the different cases which may arise when attempting to eliminate some attributes of a pattern p in order to obtain pattern p'.

4.6.3.1 Eliminating a Subset of Each Node's Attributes

This is the case when we eliminate from each return node of p only a (possibly empty) strict subset of its attributes. The important aspect is that p and p' have the same return nodes, that is: from no node of p do we remove *all* attributes.

3. Note that we have used subscripts to distinguish between the nodes of d_2 having the same labels.

Lemma 4.6.2 (Eliminating attribute subsets)**.** *Let p, p' be two structurally equivalent patterns, such that for each node $n \in p$: (i) n is a return node* iff *the corresponding node $n' \in p'$ is a return node; (ii) n has at least the attributes of n'. Let e be the algebraic expression corresponding to p. Then, $\pi_{p'}(e) \equiv p'$.*

Proof. We first show that $\pi_{p'}(p) \subseteq p$. Let $\phi : p \to d$ be a mapping from p to an XML document d, and $t \in p(d)$ be the tuple corresponding to this embedding. Let $\phi' : p \to d$ be the embedding obtained by restricting ϕ to only the attributes of p'. Clearly, ϕ' is also an embedding from p' into d; let $t' \in p'(d)$ be the tuple corresponding to ϕ'. Clearly, $\pi_{p'}(t) = t'$, where $\pi_{p'}$ is the projection removing all but p''s attributes. Generalizing this over all embeddings $\phi : p \to d$, we obtain that $\pi_{p'}(p(d)) \subseteq p'(d)$.

We now consider the opposite inclusion, that is: $p'(d) \subseteq \pi_{p'}(p(d))$. Let $t' \in p'(d)$ be a tuple produced by the embedding $\phi' : p' \to d$. Given that p and p' have the same return nodes, there is a bijection between their respective sets of embeddings into d, thus to each such embedding ϕ' corresponds exactly one embedding $\phi : p \to d$. It follows that there exists a tuple $t \in p(d)$ such that $\pi_{p'}(t) = t'$, finalizing our proof. □

As an example, consider eliminating $b.val$ from pattern p_1 depicted in Figure 4.11 to obtain p'_1. Node b remains a return node after the elimination, and $\pi_{a.val,b.ID}(p_1)$ is equivalent to p'_1. In particular, $p_1(d_1)$ and $p'_1(d_1)$ return the same two tuples (if we project out the $b.val$ column from $p_1(d_1)$).

4.6.3.2 Elimination of All Attributes of Some Nodes

This case occurs when we eliminate all attributes of some p nodes, turning them to non-return ones, that is, p' has a subset of the return nodes of p. Recall that our goal is to adjust e, the algebraic expression corresponding to p, into an expression e' corresponding to p'.

In Figure 4.11, in order to turn p_2 to p'_2 we can simply project out $c.val$, since p_2 and p'_2 give the same number of tuples when evaluated on some document (e.g., they both return 3 tuples on d_2). In other cases, though, a projection is insufficient, as we have discussed for p_3 and p'_3 in Figure 4.11, in the beginning of Section 4.6.3.

We say that a node of p is *embedding-dependent*, if by turning it to non-return and obtaining p', the embeddings of p (restricted to the remaining attributes) are the same as the ones of p' to any document d. More formally:

Definition 4.6.1 (Embedding-dependent nodes)**.** *Let p, p' be two tree patterns such that p' is obtained from p by turning a node $n \in p$ into a non-return node (removing all attributes of n). Let $\mathcal{N}_p, \mathcal{N}_{p'}$ be the sets of return nodes of p and p', respectively. Obviously, $\mathcal{N}_p = \mathcal{N}_{p'} \cup \{n\}$.*

For a given document d, let $\Phi^d_p, \Phi^d_{p'}$ be the sets of embeddings of the return nodes of p and p', respectively, into d. We denote by $\Phi^d_p|_{\mathcal{N}_{p'}}$ the restriction of Φ^d_p to $\mathcal{N}_{p'}$.

We say node n is embedding-dependent *in p,* iff *for any document d, $\Phi^d_{p'} = \Phi^d_p|_{\mathcal{N}_{p'}}$. Otherwise, we say n is* embedding-independent*.*

The following Lemma determines under which conditions n is embedding-dependent:

Lemma 4.6.3 (Embedding-dependency check)**.** *A node $n \in p$ is embedding-dependent,* iff *at least one of the following holds:*

1. *n has a return descendant n^{des} and there is a $/$-path between n and n^{des}. In this case, we say that n is* embedding-dependent on n^{des}*;*
2. *n has both a return descendant n^{des} and a return ancestor n^{anc}, and is connected to n^{anc} through a $/$-path. In this case, we say n is* embedding-dependent on n^{des} and n^{anc}.

Proof. ("If" direction) We show that if one of the two hypotheses hold, then n is embedding-dependent, that is, for any document d, we have $\Phi^d_{p'} = \Phi^d_p|_{\mathcal{N}_{p'}}$, where p' the pattern obtained from p by turning n to non-return.

Assume hypothesis *1* holds. Let d be a document and $\phi \in \Phi^d_p$ an embedding that maps (binds) n^{des} to node $n^{des}_d \in d$. Since n and n^{des} are connected with a $/$-path, starting from node $n^{des}_d \in d$ and going up the path towards the root of d, after as many steps as the length of path going from n to n^{des}, we always arrive at the same unique d node, say n_d, for any embedding ϕ that maps n^{des} to n^{des}_d. In other words, n's binding through any embedding $\phi \in \Phi^d_p$ is uniquely determined by the binding of n^{des}. Then, if we turn n to a non-return node and obtain p', the embeddings of p' will be the same as the restriction of the embeddings of p to the node set $\mathcal{N}_{p'}$, i.e., $\Phi^d_{p'} = \Phi^d_p|_{\mathcal{N}_{p'}}$.

Now assume hypothesis *2* holds. Let d be a document and $\phi \in \Phi^d_p$ an embedding that maps n^{anc} to a node $n^{anc}_d \in d$, and n^{des} to a node $n^{des}_d \in d$. Then, ϕ must map n to the *only* node that appears in d on the path going down from n^{anc}_d to n^{des}_d, after as many steps as there are $/$-edges between n^{anc} and n in p. In other words, *all* embeddings ϕ mapping n^{anc} to a fixed n^{anc}_d and n^{des} to a fixed n^{des}_d, also map n to the same node in d. In particular, this means that to any embedding $\phi' \in \Phi^d_{p'}$ corresponds exactly one embedding from $\phi \in \Phi^d_p|_{\mathcal{N}_{p'}}$, namely, the one embedding ϕ that coincides with ϕ' on its choice of n^{anc}_d and n^{des}_d. Generalizing this over all embeddings $\phi' \in \Phi^d_{p'}$, we obtain that $\Phi^d_{p'} = \Phi^d_p|_{\mathcal{N}_{p'}}$.
("Only if" direction) We show that if none of the two conditions hold, then n is embedding-independent. To that end, we build a document d such that, if we turn n into a non-return node in p', we get $\Phi^d_{p'} \neq \Phi^d_p|_{\mathcal{N}_{p'}}$. We start by building a document d isomorphic to p: for each node in p, d has a node with the same label, and for each ($/$ or $//$)-edge between two nodes in p, we add an edge between the corresponding nodes in d. For a given node $m \in p$, we denote by m^d the corresponding node in d.

The remainder of our proof is organized according to the existence or not of return ancestors and/or descendants of n, and on whether there are $/$-paths connecting them to n. With respect to the ancestors, exactly one of the following three holds: (i) n has no return ancestors; (ii) n has return ancestors but it is not connected by a $/$-path to a return ancestor; (iii) n has return ancestors and a $/$-path to at least one of them. Similarly, three cases hold for n's return descendants, allowing a total of nine possibilities. Table 4.1 depicts this space of possibilities. Hypothesis *1* of our Lemma corresponds to the bottom row, while Hypothesis *2* of the Lemma corresponds exactly to the lower two cells on the rightmost column. We now identify three cases (denoted A, B and C) which are also mutually exclusive and which, together, cover all the situations when neither of the two hypothesis hold. For each case, a different modification of the document d built out of p

above, will produce a new document demonstrating that n is not binding-independent.

	n has no return ancestor	n has return ancestors but no / path to a return ancestor	n has a /-path to some return ancestor
n has no return descendant	Case A	Case A	Case A
n has return descendants but no / path to a return descendant	Case B	Case C	Hypothesis *2*
n has a / path to some return descendant	Hypothesis *1*	Hypothesis *1*	Hypothesis *1* and Hypothesis *2*

Table 4.1: Analysis of cases for the "Only if" proof of Lemma 4.6.3.

Case A: n has no return descendant In this case, if we remove n's attributes, n is existential in the obtained pattern p'. Let m^d be the parent of n^d in d. We copy the subtree of d that is rooted at n_d, as a child of m^d. Now m^d has two children to which n can be bound through an embedding. Thus, $p(d)$ returns two tuples. However, since n is existential in p', $p'(d)$ returns only one tuple. Hence, for this particular d, $\Phi^d_{p'} \neq \Phi^d_p|_{\mathcal{N}_{p'}}$.

Case B: n has a return descendant, there is no /-path to a return descendant and n has no return ancestor The path from n to n^{des} contains at least one //-edge. Going down from n to n^{des}, let n_1 be the first node above a //-edge in p (note that n_1 and n may coincide). We alter d in the following way: we copy the path from the root of d to n^d_1 and add it as a child of n^d_1, and we move the children of n^d_1 (along with their subtrees) at the end of the newly added path. Clearly, p can now be embedded twice in the modified version of d and $p(d)$ returns two tuples. In contrast, $p'(d)$ returns only one tuple, since n is non-return in p'.

Case C: n has a return ancestor, a return descendant, and /-paths to neither Let n^{anc}, n^{des} be the return ancestor and descendant, respectively, of n. As we go up from n to n^{anc}, let n_1 be the first node below a //-edge. Similarly, let n_2 be the first node above a //-edge, on the way from n down to n^{des}. We modify d by copying the path from n_1 to n_2 and adding it as a child of n^d_1. Then, on this modified document d, $p(d)$ returns two tuples, whereas $p'(d)$ returns one tuple. □

Figure 4.11 illustrates the embedding-dependent property. In p_1, a is embedding-dependent on b (/-path to it), but b is embedding-independent (no return descendant). In p_2, b is embedding-dependent on a and c (a is return /-ancestor and c return descendant), and c is embedding-dependent on d (return /-descendant). In p_4, c is embedding-independent (no /-path neither to the return descendant not to the return ancestor).

From the proof of the above Lemma, it becomes clear that in the cases when n is embedding-independent, the embeddings of p' on any document d is a subset of the restriction on p' attributes of embeddings of p on d. This was explained also in the beginning

of this Section (Section 4.6.3): the tuples returned from p' can have smaller multiplicity than the ones of p, but not different values, which brings us to the following Lemma:

Lemma 4.6.4 (Embedding-independence)**.** *If a node n is embedding-independent, then $\Phi^d_{p'} \subseteq \Phi^d_p|_{\mathcal{N}_{p'}}$ for any document d.*

We now formally identify the cases for which we can eliminate some attributes of p through a simple projection over the equivalent expression of p:

Lemma 4.6.5 (Attr. elimination only through projection)**.** *Let p be a pattern, e the expression corresponding to p, and p' be a pattern obtained from p by removing some attributes. There exists an expression of the form $\pi(e)$ that is equivalent to p',* iff *for each node $n \in p$, some of whose attributes are eliminated in p', one of the following holds:*

1. *n remains a return node in p';*
2. *n is embedding-dependent in p' (see Definition 4.6.1).*

Proof. ("If" direction) If n remains a return node, Lemma 4.6.2 has shown that p' is equivalent to a projection over e.

Assume now that n is embedding-dependent. From Definition 4.6.1, we have that $\Phi^d_{p'} = \Phi^d_p|_{\mathcal{N}_{p'}}$ for any document d. It follows that pattern p' is equivalent to a projection over e: we only need to project out the attributes of n and then, for any document d the tuples returned by $p'(d)$ and by the projection $\pi(e)$ will be the same, since the set of embeddings of p, restricted to the attributes of p', is the same as the set of embeddings of p' on d.

("Only if" direction) We show that if none of the conditions of the lemma hold, that is, if n is non-return in p' and is not embedding-dependent either, then p' is not equivalent to a projection over e. According to Definition 4.6.1, if n is not embedding-dependent, we have $\Phi^d_{p'} \neq \Phi^d_p|_{\mathcal{N}_{p'}}$ for some document d. Thus, there exists a document d, for which the set of embeddings of p' and those of p when restricted to the attributes of p', are not the same. Therefore, p' cannot equivalent to a projection over e. □

In Figure 4.11, p'_1 is equivalent to a projection over the expression corresponding to p_1 (the b node remains return), and p'_2 can also be obtained from p_2 through a projection (the c node is embedding-dependent, due to the $/$-path from c to d). In contrast, p'_3 cannot be obtained from p_3 only through a projection.

We now turn to the cases which do not meet the conditions of Lemma 4.6.5. In such cases, a return node of p is necessarily no longer return in p' (otherwise, we fall in the situation discussed in Section 4.6.3.1) and is embedding-independent (otherwise, a projection would be sufficient to get p'). As Lemma 4.6.4 also reveals, we need to adjust the multiplicity of the tuples obtained through the projection. The only tool to do so, is by applying a duplicate elimination over the projection. However, as explained in the beginning of Section 4.6.3, there are cases, such as p_3 of Figure 4.11, when duplicate elimination does not preserve the semantics of p'_3 (in that case, we showed that depending on the values of d_1, we may get from the projection over e_3 (the expression equivalent to p_3) one or two tuples, instead of three).

The following Lemma formalizes the cases for which performing a duplicate elimination preserves the semantics of p', that is, it does not remove any desired tuples from the projection over the expression equivalent to p for some document d (as occurred in the case of p_3 explained above). Note that we first treat the case when *all* attributes are removed from some nodes of p.

Lemma 4.6.6 (Attr. elimination when dupl. elimination is necessary)**.** *Let p be a pattern, e the expression corresponding to p, and p' be a pattern obtained from p* by turning some nodes of p to non-return. *If (according to Lemma 4.6.5) p' is not equivalent to an expression $\pi(e)$, then p' is equivalent to an expression $\delta(\pi(e))$,* iff *for each return node n' of p', one of the following holds:*

1. *n' is ID-return;*
2. *n' is embedding-dependent on p' nodes that are ID-return.*

Proof. ("If" direction) First, assume that all return nodes of p' are ID-return. This means that, given a document d, $p(d)$ does not contain any duplicates, because each tuple corresponds to a unique set of bindings of the return nodes of p' to some d nodes. Since for each such d node, we also get its ID in the result of the tuple (every p' node is ID-return), $p'(d)$ does not contain any duplicates. Thus, if in $e(d)$ we project out all attributes referring to nodes of p that became non-return in p', and then perform a duplicate elimination over $\pi(e(d))$, we obtain exactly the tuples of $p'(d)$.

Assume now that there are some nodes of p' that are not ID-return, but are embedding-dependent on some other p' nodes that are ID-return (see Definition 4.6.1 and Lemma 4.6.3). Let $n^{dep} \in p'$ be such a node, and consider the case when n^{dep} depends on a single other node $n^{ind} \in p'$ (a similar discussion holds also for the situation when n^{dep} is embedding-dependent on two nodes instead of one). In this case, for a given document d, and for a given embedding $\phi : p' \rightarrow d$, $\phi(n^{ind})$ determines the d node in which n^{dep} is mapped. Thus, based on the unique set of bindings of the ID-return p' nodes and the fact that the remaining return nodes are embedding-dependent on them, there will be no duplicates in $p'(d)$. It follows that, as explained for the case when all return nodes are ID-return, the expression $\delta(\pi(e))$ is equivalent to p'.

("Only if" direction) Let n' be a return, but not ID-return, node in p, which is embedding-independent. It is easy to create a document (such as the ones described in the proof of Lemma 4.6.3), such that for the same set of bindings of the ID-return nodes of p', n' can be bound to two different d nodes, say n_{d1}, n_{d2}, leading to two tuples. If n' is val-return and $n_{d1}.val = n_{d2}.val$, then the two tuples are the same (but both are included in the result of $p'(d)$ because they correspond to unique sets of bindings of the return p' nodes), and performing a duplicate elimination on $\pi(e)$ will lead to a different semantics than p'; in particular, they will differ on the document we constructed. Thus, no expression of the form $\delta(\pi(e))$ can be equivalent to p', because the duplicate elimination will remove one of the two afore-mentioned tuples. The same can be shown if n' is $cont$-return. □

For instance, in Figure 4.11, to obtain p'_3 from p_3, a duplicate elimination is needed (c becomes non-return and there is no $/$-connected ID-return descendant). However, the conditions of Lemma 4.6.6 are not met, hence, we cannot obtain p'_3 from p_3. On the

contrary, we can project out $c.val$ from p_4 and apply a duplicate elimination (b and d are ID-return and a is embedding-dependent on b).

Consider now pattern p_5 in Figure 4.11, and assume we want to remove some of its attributes and obtain pattern p_6. Let e_5 be the equivalent expression to p_5. We first apply a projection over p_5, that is, $\pi_{c.val}$ to remove the attributes not present in p_6. Clearly, a duplicate elimination is needed on top of the projection (b that was turned to non-return is embedding-independent in p_6). However, we cannot apply Lemma 4.6.6 on p_6 to check if a duplicate elimination can be performed, because, although we removed part of c's attributes, we did not remove all of them, as the Lemma requires. Moreover, notice that the rest of the requirements of Lemma 4.6.6 are not met either, and according to that, we cannot perform a duplicate elimination over the projection and preserve p_5 semantics. Nevertheless, we can obtain p_6 by first obtaining p'_5. In fact, p_6 is equivalent to the expression $\pi_{c.ID,c.val}(\delta(\pi_{c.val}(e)))$, which brings us to the following Lemma:

Lemma 4.6.7. *Let p be a pattern, e the expression corresponding to p, and p' be a pattern obtained from p removing some of its attributes. Let p'' be a pattern obtained from p by (i) keeping only the attributes of p'; (ii) for each return node of p', if the corresponding node in p had an ID, add that ID attribute in p'.*

Then, p' is equivalent to the expression $\pi_{p'}(\delta(\pi_{p''}(e)))$, iff *$p''$ is equivalent to $\delta(\pi_{p''}(e))$ (where we denoted $\pi_{p_x}(e)$ a projection keeping exactly the attributes of p_x).*

Proof. We start by clarifying the role of p''. This pattern is an "intermediary version" between p and p', having at most the attributes of the former and at least the attributes of the latter. More specifically, p'' preserves IDs for all p nodes that are still return in p', while being restricted to p''s attributes otherwise. For instance, if $p = //a_{ID,val}//b_{val}[//c_{ID,val}]//d_{val}$ and $p' = //a//b[//c_{val}]//d_{val}$, then $p'' = //a//b[//c_{ID,val}]//d_{val}$.

The "if" direction is straightforward: if $p'' \equiv e'' = \delta(\pi_{p''}(e))$, then it follows from Lemma 4.6.5 that $p' \equiv \pi_{p'}(e'') = \pi_{p'}(\delta(\pi_{p''}(e)))$, since p'' and p' have the same return nodes.

For the "only-if" direction, we show that if $p'' \not\equiv \delta(\pi_{p''}(e))$, then $p' \not\equiv \pi_{p'}(\delta(\pi_{p''}(e)))$. Since $p'' \not\equiv \delta(\pi_{p''}(e))$, we have one of the following:

1. $p'' \equiv \pi_{p''}(e)$ (Lemma 4.6.5). Then according to the same Lemma, $p' \equiv \pi_{p'}(\pi_{p''}(e))$, thus, indeed $p' \not\equiv \pi_{p'}(\delta(\pi_{p''}(e)))$.
2. p'' cannot be equivalent to an expression over e even after duplicate elimination (Lemma 4.6.6), thus, the semantics of p'' are not preserved through $\delta(\pi_{p'}(e))$. If this is the case, we cannot perform a projection over $\delta(\pi_{p'}(e))$ and preserve the semantics of p' that has the same return nodes as p'' (Lemma 4.6.5). Hence, we have $p' \not\equiv \pi_{p'}(\delta(\pi_{p''}(e)))$. Moreover, p' cannot be equivalent to $\delta(\pi_{p'}(e))$: if p'' cannot be equivalent to $\delta(\pi_{p''}(e))$ that has even more ID-return nodes than p (Lemma 4.6.6 has shown that ID-return nodes can only help in applying duplicate elimination while preserving the semantics of the target pattern), then definitely duplicate elimination cannot be successfully applied over $\pi_{p''}(e)$ to obtain p'.

□

Algorithm 4: Attribute Elimination (**AttElim**)

Input : Tree patterns p, p', with $p \overset{s}{\equiv} p'$ and p' having a subset of p's attributes; expression $e \equiv p$
Output: Expression $e' = \alpha(e)$ such that $e' \equiv p'$

```
//Check if p' ≡ π(e) (Lemma 4.6.5)
1  only_projection ← true
2  foreach node n ∈ p s.t. S_at(n,p) ≠ S_at(n,p') do
3      if S_at(n,p') = 0 and n embedding-independent in p' then
4          only_projection ← false
5  if only_projection = true then return π_{p'}(e)
   //Check if p' ≡ δ(π(e)) (Lemma 4.6.6) or p' ≡ π(δ(π(e)))
     (Lemma 4.6.7)
6  Build p'' from p', adding ID to node n ∈ p'' if n had an ID in p and S_at(n,p') ≠ 0
7  foreach node n ∈ p'' s.t. S_at(n,p'') ≠ 0 do
8      if n !ID-return and !embedding-dependent on ID-return nodes then
9          return null
10 if p' ≡ p'' then return δ(π_{p''}(e))
11 else return π_{p'}(δ(π_{p''}(e)))
```

4.6.3.3 Attribute Elimination Algorithm

We now present Algorithm **AttElim** (Algorithm 5), which takes as inputs two patterns p and p', such that $p \overset{s}{\equiv} p'$ (see Definition 4.4.5), with p' having a subset of p's attributes, as well as the expression $e \equiv p$. **AttElim** attempts to eliminate these additional attributes in order to obtain p' through an algebraic expression e' built over e. If the algorithm succeeds, it returns e'. However, it may fail to find such an algebraic expression.

In the algorithm, $S_{at}(n, p)$ is the node signature of a node $n \in p$ (see Definition 4.4.1). Since $p \overset{s}{\equiv} p'$, if n is a node of p, then p' also has a node n of the same label. However, if $S_{at}(n, p) \subset S_{at}(n, p')$, then some attributes of $n \in p$ should be eliminated.

First, we check if p' is equivalent to a projection over e, that is, an expression of the form $\pi(e)$, according to Lemma 4.6.5 (lines 1-4).

If this is not the case, a duplicate elimination is also needed on top of the projection over e. To this end, in the rest of the algorithm, we check if a duplicate elimination can be successfully applied, while preserving the semantics of p', leading to an expression $\delta(\pi(e))$ (Lemma 4.6.6) or an expression $\pi(\delta(\pi(e)))$ (Lemma 4.6.7). Otherwise the algorithm fails, meaning that p' cannot be obtained through an expression over e.

Proposition 4.6.8 (Completeness of **AttElim**). *Algorithm **AttElim** is complete, i.e., given two tree patterns p, p' as input, such that $p \overset{s}{\equiv} p'$ and p' having a subset of p's attributes, if p' is equivalent to an algebraic expression over e (the expression equivalent to p), **AttElim** will find one such expression.*

The completeness of **AttElim** follows readily from Lemmas 4.6.5, 4.6.6 and 4.6.7,

which capture all cases in which projection and duplicate elimination can be used to algebraically obtain p' from p.

4.6.4 Node Unification (NodeUnif)

In this Section we present the node unification transformation which, given a (DAG or tree) pattern, aims at *unifying* (or collapsing) some of its nodes, which has the net effect of turning the pattern into one with fewer nodes. In our query rewriting context, the goal is to unify nodes of the pattern that are mapped to the same query node: in order to arrive to an expression equivalent to the query, the mapping from the pattern that is equivalent to the rewriting expression, to the query should be bijective (see Theorem 4.4.3).

Formally, we are given as input a DAG pattern dp along with its corresponding algebraic expression e, and a structural embedding $\phi : dp \xrightarrow{s} p$ from dp into a tree pattern p (in our context, the role of p will be played by a tree pattern extracted from the query that we seek to rewrite). We say two nodes $n_1, n_2 \in dp$ are *unification candidates* whenever $\phi(n_1) = \phi(n_2)$, that is, whenever the two nodes are mapped to the same p nodes. The *unification* of n_1 and n_2 is defined as the process that substitutes both n_1 and n_2 in dp, by a single node n_u, (i) having as attributes the union of the attributes of n_1 and n_2, (ii) as direct ancestors in the pattern, the union of the ancestors of n_1 and n_2, and (iii) as descendants, the union of their descendants. Node unification on a DAG pattern may lead to either a tree or a DAG pattern, and similarly, from a tree pattern, unification may produce either a tree or a DAG. That is why, for most generality, we define it on DAG patterns. Note that the output pattern of node unification can be also structurally embedded to p through ϕ, if we also substitute the mapping of n_1 and n_2 to a node n_p of p, by the mapping of n_u to the same n_p node.

Given dp, e, ϕ, and a pair of unification candidates, transformation **NodeUnif-1** attempts to unify the unification candidates, and returns the modified versions of dp, e and ϕ. In order to unify two nodes n_1 and n_2 through an algebraic expression, both of them need to be ID-return nodes. Then, we add a selection over e, imposing that $n_1.ID = n_2.ID$. In the resulting pattern, n_1 and n_2 are unified into node n_u. If n_1 and n_2 have a common direct descendant, n_u will be connected to it through a $//$-edge if both n_1 and n_2 were connected to it with such an edge, otherwise through a $/$-edge.

For instance, in Figure 4.12, both d nodes of dp are ID-return and are mapped to the same p node. To this end, we apply a selection on e (let $e' = \sigma(e)$ be the corresponding expression) unifying the two d nodes, and obtain dp'.

Confluence and order-independence For a given dp, e and ϕ, several unification candidate pairs may exist, in which case **NodeUnif-1** may be applied repeatedly, leading to a chain of transformations of the form:

$$(dp, e, \phi) \xrightarrow{\text{NodeUnif-1}} (dp_1, e_1, \phi_1) \xrightarrow{\text{NodeUnif-1}} (dp_2, e_2, \phi_2) \ldots \xrightarrow{\text{NodeUnif-1}} (dp_k, e_k, \phi_k)$$

It turns out that whenever several **NodeUnif-1** transformation can apply, the order in which they are applied does not change the final result. On the algebra side, this

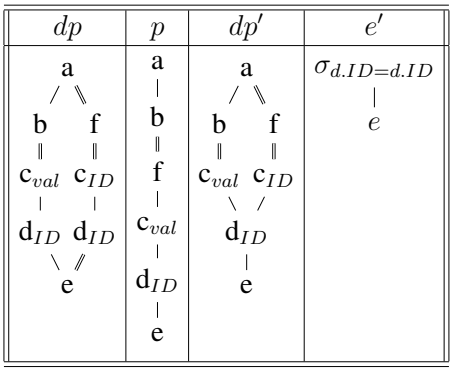


Figure 4.12: Node unification.

is confirmed by the intuition that selections are commutative; on the pattern side, the corresponding observation is that unifying two nodes never prevents the unification of another pair of nodes.

This enables us to define the **NodeUnif** transformation which, given a (d, p, e) triple, identifies all unification candidates and repeatedly applies **NodeUnif-1** until there is no unification candidate pair left.

Proposition 4.6.9 (**NodeUnif** completeness)**.** ***NodeUnif*** *is complete, i.e., given a pattern dp that is structurally embedded to a tree pattern p, and the expression e that is equivalent to dp, if there is an algebraic expression over e that can unify the nodes of dp mapped to the same p node, then* ***NodeUnif*** *will find such an expression.*

Proof. Given all operators of our algebra, adding an ID selection is the only way to algebraically impose that two nodes are the same. Moreover, the above observation about the order of applying node unification entails that **NodeUnif** will output a pattern, in which no more nodes can be algebraically unified. □

Note that we have presented a different way of unifying nodes, through the **DAG2Tree** operation, in Section 4.4.5. However, **DAG2Tree** does not apply algebraic operations. Instead, it uses properties of DAG patterns, namely the interleavings, in an attempt to transform a DAG pattern to a tree. During **DAG2Tree**, two nodes that have the same immediate common descendant and are connected to it through a $/$-edge, may be unified; the two nodes do not need to be ID-return. For instance, in Figure 4.12, the two c nodes cannot be unified through **NodeUnif**, but they can be unified through **DAG2Tree**.

4.6.5 Structural Refinement (StructRef)

Our last algebraic pattern transformation is structural refinement, which takes as input a DAG pattern dp (together with its equivalent algebraic expression e) and a tree pattern p, such that there exists $\phi : dp \xrightarrow{s} p$. Again, in practice, the role of p is played by a tree pattern obtained from the query we seek to rewrite. Structural refinement attempts to add some structural relationships not originally present on dp, but which are present in p.

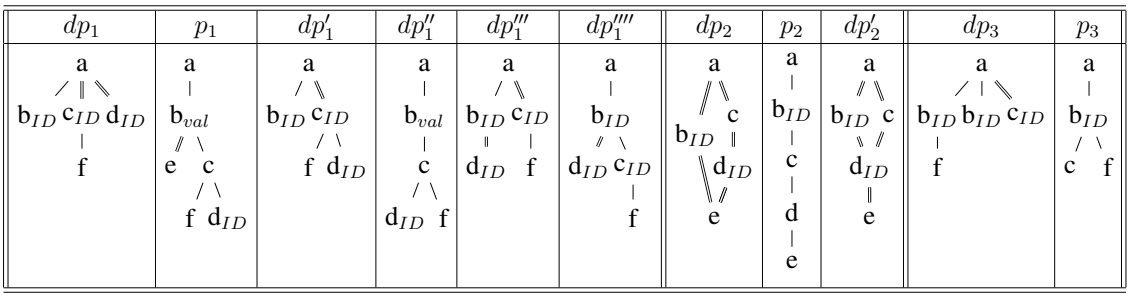


Figure 4.13: Structural refinement.

In particular, consider two nodes $n_1, n_2 \in dp$, such that there exists a path (of one or more edges) going down from $\phi(n_1)$ to $\phi(n_2) \in p$, whereas no path goes down from n_1 to n_2 in dp. Assume that n_1 is the *only* ID-return dp node mapping to $\phi(n_1)$, in other words, for any ID-return $n \in dp$ such that $n \neq n_1$, $\phi(n) \neq \phi(n_1)$ holds. We define the transformation **StructRef-1**, such that n_1, n_2 are both ID-return, it can algebraically add one of the following relationships:

1. if there is a $//$-edge or a path (consisting of both $/$- and $//$-edges) between $\phi(n_1)$ and $\phi(n_2)$ in p, we connect n_2 through a $//$-edge to n_1 and obtain a new DAG pattern dp'. The algebraic counterpart of this operation is to add a selection of the form $n_1.ID \ll n_2.ID$ on top of e;
2. if $\phi(n_2)$ is a $/$-child of $\phi(n_1)$, we connect n_2 through a $/$-edge to n_1 and obtain a new DAG pattern dp'. To keep the algebraic expression aligned with the modified pattern, we add a selection of the form $n_1.ID \prec n_2.ID$ on top of e.

The reason for the constraint that ϕ maps only n_1 to $\phi(n_1)$ and no other ID-return dp node, is to ensure **StructRef-1** is deterministic (had there been more nodes like n_1, it would not be clear to which of them to connect n_2). Moreover, note that in all cases there exists $\phi' : dp' \xrightarrow{s} p$, mapping the dp' nodes to the same p nodes that ϕ mapped the nodes of dp.

For instance, consider the patterns in Figure 4.13. We have $dp_1 \xrightarrow{s} p_1$, and we can impose the selection with predicate $c.ID \prec d.ID$ (which is present in p_1 but not in dp_1) and obtain pattern dp_1. Then, we can apply $b.ID \prec c.ID$ and obtain dp''_1. Observe that there are no more relationships from in p_1 that could be added to dp_1. Likewise, we have $dp_2 \xrightarrow{s} p_2$, and we can impose the relationship $b.ID \prec d.ID$ to obtain the DAG pattern dp'_2. However, in dp_3 we cannot impose the relationship that b is a parent of c (which appears in p_3), because there are two ID-return b nodes in dp_3 mapped to the same p_3 node.

In a very similar way to **NodeUnif-1**, **StructRef-1** can be applied repeatedly, and moreover, the order in which a sequence of **StructRef-1** transformations are applied does not change the result. The intuition for the proof is again that the order of selection operators does not impact the algebraic semantics, and no application of **StructRef-1** precludes another one. Thus, we define the operation **StructRef** which, given as input a DAG pattern dp and a tree pattern p such that $dp \xrightarrow{s} p$, and an expression e equivalent to

dp, repeatedly applies **StructRef-1** on dp, e and ϕ until no relation from p has been left to apply on dp.

For instance, consider patterns dp_1, p_1 in Figure 4.13 that are given as input in **StructRef**. Regardless the order in which relationships are added, the output pattern should be dp_1''. An order could be to start from dp_1, then obtain dp_1' and then dp_1'', as explained above. However, one could also start from dp_1, then obtain dp_1''' (b ancestor of d), then obtain dp_1'''' (b is parent of c), and finally obtain dp_1'' (c parent of d).

Proposition 4.6.10 (**StructRef** completeness)**.** ***StructRef*** *is complete, i.e., given a pattern dp that is structurally embedded to a tree pattern p, and the expression e that is equivalent to dp, if there is an algebraic expression over e that can impose additional structural relationships to dp that are present in p, then* ***StructRef*** *will find such an expression.*

Proof. Given our algebra, the selection is the only operator that can be used to algebraically impose a new structural relationship on a pattern. Furthermore, since we showed that the order in which we impose the additional relationships does not alter the result of **StructRef**, it is guaranteed that it will output a pattern in which no more relationships can be imposed algebraically, together with the algebraic expression. □

We come back to our last example for an interesting remark. On the two **StructRef** paths from dp_1 to dp_1'' (different orderings of the individual refinement steps), the selection predicates in the first case were $b.ID \prec c.ID$, $c.ID \prec d.ID$, whereas in the second they were $b.ID \prec c.ID$, $c.ID \prec d.ID$ and $b.ID \not\prec c.ID$. Clearly, the last predicate in the second case is not necessary (it is covered by the other two).

A simple efficient algorithm can be devised for implementing **StructRef** while avoiding such unnecessary predicates. We visit each node $n \in dp$ and mark node $\phi(n) \in p$ whenever n is ID-return and there is no other dp node mapping to $\phi(n)$. Then, we perform a top down traversal of p and for each marked node n_p, we apply in dp the relationship that n_p has with its closest marked ancestor.

Finally, observe that **DAG2Tree** (see Section 4.4.5) can also impose structural relationships that are not present in the initial pattern, but in this Section we discuss only algebraic transformations.

4.7 Tree Pattern Queries Rewriting Details

We have given an overview of our tree pattern rewriting algorithm **TPR** in Section 4.5. The algorithm relied on a set of functions: **FilterView**, **TPTransform**, **PickPair** and **TPJoin**. Each of these functions is relatively complex, thus in Section 4.5 we only gave some intuition for their roles.

In this Section, we spell out all the specifics of algorithm **TPR** which were not given in Section 4.5. Section 4.7.1 details the functions mentioned above, whereas Section 4.7.2 tackles the termination and completeness of the **TPR** algorithm.

4.7.1 Rewriting Algorithm Functions

In this Section we detail the functions participating in the TPR Algorithm 4 that were briefly presented in Section 4.5.

4.7.1.1 View Filtering (Function ViewFilter)

Given the set of tree pattern views $\mathcal{V}$, **ViewFilter** discards some views that cannot participate in an equivalent rewriting of the query q. This function is called by Algorithm **TPR** before actually attempting to build rewritings. The purpose is to reduce the number of views given as input to the rewriting process, and thus reduce the complexity of rewriting q.

Function **ViewFilter** discards a view $v \in \mathcal{V}$, if one of the following holds:

1. There is no structural embedding (Definition 4.4.3) from v into q.
2. The following conditions hold: (i) there is no structural embedding from ϕ into the query such that all the query nodes are target nodes of the embedding; (ii) any transformation of v by expanding v's pattern through nav operators as described in Section 4.6.2 leads to a pattern v' which does not satisfy condition (i) above, either; (iii) no node of v is ID-return.
3. The following two conditions hold: (i) for any structural embedding from v to q, there exist some nodes $n_v^1, n_v^2, \ldots, n_v^k \in v$ (not necessarily the same nodes for each ϕ), such that $\phi(n_v^1) = \phi(n_v^2) = \ldots = \phi(n_v^k) = n_q$ for some query node n_q, and the union of the attributes of $n_v^1, n_v^2, \ldots, n_v^k$ is not a superset of n_q's attributes; (ii) no node of v is ID-return.

Proposition 4.7.1 (Soundness of **ViewFilter**). ***ViewFilter*** *is sound, i.e., it does not discard views that could be used in a rewriting of the query.*

Proof. We show that none of the views that are discarded by **ViewFilter** could lead to a rewriting of query q.

Condition *1* requires a structural embedding $\phi : v \xrightarrow{s} q$ in order to keep v. It has been previously shown in the case of XPath (single return node) queries and views, that such an embedding is a necessary condition in order for a view to be able to participate in a query rewriting [TYÖ+08], and the proof for our more general language is very similar. Intuitively, when v cannot be embedded in the query, this means that there is some condition (edge) in v such that a similar structural condition does not hold in the query. However, none of our algebraic operators can *remove* (relax) structural or value constraints, but only possibly *add* constraints (e.g., add a value selection predicate, add an equality over the IDs of two nodes, etc.). Therefore, the rewriting process will not be able to remove the view constraint preventing a structural embedding into the query, and the view could participate at most in a partial (not equivalent) query rewriting. This is why the view can be removed.

Conditions *2* and *3* consider cases when a view is not sufficient on its own (does not store all required information) to answer q, either in terms of structure (condition *2*) or in terms of return attributes (condition *3*). In such cases, v needs to have some ID-return

nodes, so that it can be joined with other views covering the query nodes (or attributes) that v does not. If v does not have such join-enabling IDs, it can be safely discarded. □

4.7.1.2 Tree Pattern Transformations (Function TPTransform)

In this Section we describe the function **TPTransform**, which applies a set of algebraic transformations on a given pattern in order to add on it some conditions from the query. **TPTransform** is outlined in Algorithm 6. It takes as input a triple (tree pattern p, equivalent algebraic expression e, embedding $\phi : p \xrightarrow{s} q$) as well as a query q. First (lines 3-6), **TPTransform** applies all algebraic transformations presented in Section 4.6 until no more new (pattern, expression, embedding) triples can be obtained. As already explained, our transformations may yield a DAG instead of a tree pattern. However, we are interested only in LDQT rewritings (see Section 4.5.2), thus, all our partial rewritings need to be tree patterns. To that end, for each obtained pattern, we apply the **DAG2Tree** operation (Section 4.4.5) which seeks to transform the obtained pattern into a tree pattern. If **DAG2Tree** does not succeed, we can safely discard p. Finally, we minimize the remaining tree patterns (see Section 4.4.2) and output them. The minimization is needed, since the subsequent steps (including the tree pattern equivalence check) operate over minimized tree patterns.

As we showed in Section 4.6, each of our transformations takes as input a (DAG or tree) pattern p and a tree pattern p', and tries to impose as many restrictions as possible from p' to p. For instance, **ValPredAd** will attempt to add on p as many value predicates of p' as possible.

When a sequence of successive transformations can be applied, it may be the case that the state obtained after applying *all* of them cannot lead to a rewriting, while an intermediary state reached by applying only a subset of the transformations may have allowed it. For instance, consider the pattern p_1 and the query q_1 in Figure 4.14. If we apply **AttElim**(p_1, q_1), the $b.ID$ will be eliminated from p_1. However, if the tree pattern $p'_1 = b_{ID,val}/c$ is available, we can join p_1 with p'_1 and get a rewriting for q_1.

To avoid missing rewritings this way, during **TPTransform** we do not give as second input to the transformation functions the query as such, but various modified versions of it (patterns structurally equivalent to q but having subsets of its attributes, subtrees of q, etc.), so that we generate all such intermediary states and preserve completeness.

Moreover, notice that two transformations τ_1, τ_2 may alternate on a transformation path, that is, one may apply τ_1, then τ_2, then τ_1 again etc. For instance, consider pattern p_2 and query q_2 in Figure 4.14. We first apply a **StructRef** and obtain p'_2. In p'_2 we cannot impose the restriction that b is a parent of c (although it appears in q_2), because we have two ID-return b nodes (see Section 4.6.5). However, we can apply **NodeUnif**, obtain p''_2 in which the two b nodes are unified, and then apply on p''_2 **StructRef** to obtain a pattern equivalent to the query.

Proposition 4.7.2 (Completeness of **TPTransform**). *Given a tree pattern p, an algebraic expression e equivalent to p, a query q, and an embedding $\phi : p \xrightarrow{s} q$, **TPTransform** outputs all (tree patterns p', expression e', structural embedding ϕ' embedding $p' \xrightarrow{s} q$) triples such that:*

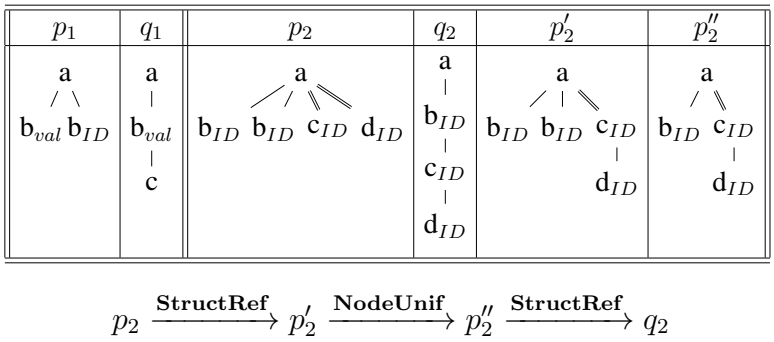


Figure 4.14: Ordering of transformations in **TPTransform**.

Algorithm 5: Tree Pattern Transformations (**TPTransform**)

Input : Query q, tree pattern p, algebraic expression e equivalent to p, embedding $\phi : p \xrightarrow{s} q$

Output: Set S_{out} of triples (p_i, e_i, ϕ_i), such that p_i a tree pattern obtained through expression e_i built upon e, $p \xrightarrow{s} p' \xrightarrow{s} q$ (ϕ_i is the embedding of p_i to q), and $S_{at}(p') \subseteq S_{at}(p)$

//set of algebraic transformations

$\mathcal{T} = \{\textbf{ValPredAd}, \textbf{Nav}, \textbf{AttElim}, \textbf{NodeUnif}, \textbf{StructRef}\}$

$S_{out} = \{(p, e, \phi)\}$

//Apply all transformations to obtain new patterns

while *new triples are added to* S_{out} **do**

  foreach *pattern* $p_c \in S_{out}$ **do**

    apply a transformation $\tau \in \mathcal{T}$ on p_c

    add the new (p_n, e_n, ϕ_n) to S_{out}

//Transform DAGs to trees and discard those that failed

foreach *pattern* $p_c \in S_{out}$ **do**

  $p_c \leftarrow$ **DAG2Tree**(p_c)

  if $(p_c) = null$ **then** remove p_c from S_{out}

  else $p_c \leftarrow$ **Minimize**(p_c)

return S_{out}

1. *e' is an algebraic expression built on top of e, that is, $e' = \alpha(e)$ for some combination α of operators from our algebra*[4]*,*
2. *we have $p \xrightarrow{s} p'$, and*
3. *for each node $n_i \in p$ such that $\phi(n_i) = n'$, we have $S_{at}(n', p') \subseteq \cup_{n_i \in p}(S_{at}(n_i, p))$, that is, the union of attributes of the p nodes that are mapped to the same node $n' \in p'$ is a superset of n''s attributes.*

4. Note that join operators are not allowed to participate in α, since we have not considered the joining of patterns so far.

Proof. Let (p', e', ϕ') be a triple satisfying the above three characteristics. We show that p' can be obtained by **TPTransform**.

Since there exists $\phi : p \xrightarrow{s} p'$, the structures of p and p' may only differ in the following ways (see Section 4.4.1):

- p' may have some nodes to which no p node corresponds, that is, there may exist a node $n' \in p'$ such that for any node $n \in p$, $\phi_p(n) \neq n'$;
- some nodes of p' may have some value predicates that do not appear in the corresponding nodes of p, that is: for some nodes $n \in p$ and $\phi_p(n) \in p'$, $\phi_p(n)$ carries a predicate not present on n (the presence of the embedding ensures that if n has a predicate, $\phi_p(n)$ also has it);
- a //-edge (n_1, n_2) in p may be embedded through ϕ_p into a longer path from $\phi_p(n_1)$ to $\phi_p(n_2)$ in p';
- two nodes in p may be embedded by ϕ_p into the same p' node.

Moreover, p and p' may differ in their respective attributes, to the extent allowed by hypothesis *3*.

Let $\mathcal{P}$ be the path of algebraic operators leading from p to $\alpha(p) \equiv p'$. The only operators that may appear in $\mathcal{P}$ are: σ (on value or ID conditions), nav, δ and π. Importantly, thanks to the commutativity properties of these operators, there exists a path $\mathcal{P}'$ leading from p to p', and of the form:

$$nav^* \ \sigma^* \ (\pi \mid \delta)^*$$

More specifically, let e_1 be the algebraic expression obtained after the $nav^* \ \sigma^*$ prefix of $\mathcal{P}'$. It is easy to see that e_1 corresponds to a pattern p_1: the nav and σ algebraic operators, when given a (tree or DAG) pattern as input, are guaranteed to also output a pattern. From the definition of our **Nav**, **ValPredAd**, **NodeUnif** and **StructRef** transformations, it follows that we are able to build a *transformation* path going from p to p_1 (and its equivalent expression e_1).

Remains then to be shown that there exists a transformation path leading from p_1 to p'. Let $\mathcal{P}_2$ be the second part of $\mathcal{P}$, from p_1 to p'; recall that $\mathcal{P}_2$ is of the form $(\delta \mid \pi)^*$. Let att denote the attributes of e_1 which are kept after applying all the projections of $\mathcal{P}_2$. We build the pattern p_2 that is structurally equivalent to p_1 and has exactly the attributes att. We invoke the **AttElim** algorithm to find a way to compute p_2 out of p_1. Proposition 4.6.8 has established that if an algebraic expression can compute p_2 out of p_1, **AttElim** will find it and output an expression of the form $e_2 = (\pi \mid \delta)^*(e_1)$. If p_2 is not already p', then p' can be obtained from p_2 by applying our transformation **DAG2Tree**.

□

Proposition 4.7.3 (Termination of **TPTransform**)**.** *Given a tree pattern p, an algebraic expression e equivalent to p, a query q, and an embedding $\phi : p \xrightarrow{s} q$, **TPTransform** terminates.*

Proof. By definition, **TPTransform** produces tree patterns p' that are more restrictive than p and less than q, since we have $p \xrightarrow{s} p' \xrightarrow{s} q$. By exhaustively applying our transformations, we reach a point when we cannot further restrict the obtained tree patterns without adding (structural or value) constraints that are not present in the query. Thus, the

Algorithm 6: Tree Pattern Joining (**TPJoin**)

Input : Query q, triples (p_1, e_1, ϕ_1),(p_2, e_2, ϕ_2), where e_1,e_2 two partial rewriting expressions, p_1, p_2 their tree patterns, ϕ_1, ϕ_2 their structural embeddings to q

Output: Set S_{out} of triples (p, e, ϕ), where p the result of joining p_1 with p_2, e the expression to obtain p, ϕ the embedding from p to q

```
1  S_out ← ∅
2  foreach ID-return node n_1 ∈ p_1, n_2 ∈ p_2 do
3      p = null
4      if φ_1(n_1) = φ_2(n_2) then
5          e ← e_1 ⋈_{n_1.ID=n_2.ID} e_2
6          create DAG p from p_1, p_2 by unifying n_1 with n_2
7      else if φ_1(n_1) parent of φ_2(n_2) then
8          e ← e_1 ⋈_{n_1.ID≺n_2.ID} e_2
9          create DAG p from p_1, p_2 by adding edge n_1/n_2
10     else if φ_1(n_1) ancestor of φ_2(n_2) then
11         e ← e_1 ⋈_{n_1.ID≪n_2.ID} e_2
12         create DAG p from p_1, p_2 by adding edge n_1//n_2
13     if p ≠ null then
14         φ = φ_1 ∪ φ_2
15         add (p, e, φ) to S_out
16 return S_out
```

procedure of applying structural restrictions on p to get new patterns terminates. Moreover, removing attributes from the obtained patterns (through **AttElim**) also terminates (in the worst case, when no more attributes are left in the pattern).

□

Optimizations on TPTransform We now make some observations that can lead to a more efficient version of **TPTransform**. Formally defining these observations and proving that they do not compromise the completeness of **TPTransform** is left as future work.

It is easy to show that **ValPredAd** can be exhaustively applied only once for every (pattern, expression, embedding) in the beginning of **TPTransform**. Moreover, we can show that **NodeUnif** and **StructRef** can be exhaustively applied on the versions of the patterns obtained after applying **AttElim** and **Nav**, and in particular **NodeUnif** can be applied after **StructRef** (based on the fact that **NodeUnif** can enable some **StructRef** applications that were not possible before). We are currently investigating also other possible ways to reduce the number of created intermediary states in order to enhance the performance of **TPTransform**.

4.7.1.3 Joining Two Partial Rewritings (Function TPJoin)

The **TPJoin** function is responsible for combining two partial LDQT rewritings, each based on one or more views, in order to obtain new partial LDQT rewritings. The combination is done via a join (either by ID equality, or by the structural condition $\prec$ or $\prec\!\!\prec$).

TPJoin is outlined in Algorithm 7. It takes as input two triples of the form (tree pattern p_i, algebraic expression e_i, embedding $\phi_i : p_i \xrightarrow{s} q$), as well as query q. It forms all pairs (n_1, n_2) of ID-return nodes $n_1 \in p_1$ and $n_2 \in p_2$, and attempts to apply an equi- or structural-join on them. More specifically:

- If both nodes are mapped to the same q node, that is, $\phi_1(n_1) = \phi_2(n_2)$, an equi-join is attempted.
- If $\phi_1(n_1)$ and $\phi_2(n_2)$ are connected through a path in q, a structural join is attempted: (i) on the condition $n_1 \prec n_2$, if $\phi(n_1)$ is a $/$-parent of $\phi(n_2)$ in q, (ii) on the condition $\phi(n_1) \prec\!\!\prec \phi(n_2)$, if $\phi(n_1)$ is a $//$-parent or an ancestor of $\phi(n_2)$.
- If none of the above holds, $\phi_1(n_1)$ and $\phi_2(n_2)$ do not belong to the same path in q, and thus cannot be combined through a join.

The resulting pattern is, in the general case, a DAG pattern p. **TPJoin** outputs the pattern together with the equivalent algebraic expression e and the embedding $\phi : p \xrightarrow{s} q$ (resulting from the union of ϕ_1 with ϕ_2).

Notice that while pattern p may be a DAG and not a tree pattern, Proposition 4.5.2 has stated that we only need to keep those partial rewritings that are LDQTs, in particular, that are equivalent to some subtree of the query. For instance, let $v_1 = //a//b_{ID}$ and $v_2 = //c//b_{ID}$, $e_1 = scan(v_1)$ and $e_2 = scan(v_2)$, $q = //a//c//b_{ID}$, and assume we build the partial rewriting $e_3 = e_1 \bowtie_{b.ID} e_2$. In the resulting DAG pattern, the b node has a $//$-parent labeled a and another $//$-parent labeled c. Clearly, this DAG pattern is not equivalent to any tree pattern, and in particular to any part of the query. Thus, Proposition 4.5.2 allows us to discard the partial rewriting e_3.

A second important observation is that in some cases where the result of a join is not an LDQT, applying some transformations may transform it into an LDQT one. For example, let $v'_1 = //a_{ID}//b_{ID}$ and $v'_2 = //c_{ID}//b_{ID}$, and let $q = //a//c//b_{ID}$. As in the previous example, let $e'_1 = scan(v_1)$, $e'_2 = scan(v_2)$ and let the plan $p = e'_1 \bowtie_{b.ID} e'_2$. In a way similar to the above example, the plan p has no equivalent tree pattern. However, if node IDs are structural, one can build $p' = \sigma_{a.ID \prec\!\!\prec c.ID}(p)$, which is an LDQT rewriting for q. The selection is inspired by the query, which specifies that a should be an ancestor of c.

In order to exploit all such opportunities of turning a DAG into a tree patter, with or without adding algebraic operators, the rewriting algorithm (**TPR**) applies **TPTransform** on each result returned by **TPJoin** (line 17).

Optimizations on TPJoin Instead of performing all possible joins between the ID-return nodes of two patterns, it can be shown that we can attempt a much smaller number of joins, without losing any rewritings. For instance, when performing an equi-join between two patterns, we do not need to consider all possible equi-joins between the two nodes of the patterns, because many of them will end up producing the same patterns multiple times after the application of **TPTransform** (e.g., due to **NodeUnif**, which also imposes

ID-equality predicates between nodes, the join result will be the same regardless of the pair of nobes we choose to perform the join). Formalizing this remark and proving that it preserves completeness is left as future work.

4.7.1.4 Join Enumeration Strategies (Function PickPair)

In Algorithm **TPR** (section 4.5), the function **PickPair** dictates the order in which the partial rewritings will be chosen to be combined by **TPJoin**. In this Section we discuss **PickPair**'s inner workings.

PickPair only combines a pair of rewritings such that one of them is built from a single view. In other words, **PickPair** builds left-deep join trees. We do this in order to (i) avoid building all the bushy join trees equivalent to a given left-deep join tree, since the choice of the join order rightfully belongs to the optimizer and should not be made during rewriting, and (ii) grow our rewritings by only one view at a time, so as to be able to check immediately after the join, if the newly added view brings some information to the rewriting. If it does not, then the new join is non-minimal, and it is discarded directly.

We describe below three possible strategies for **PickPair**. Observe that other strategies could also be applied.

Naïve dynamic programming (NDP) first attempts all joins of a partial rewriting built on one view, with another similar partial rewriting built on two views. Then, it attempts all joins between partial rewritings of two views, and partial rewritings of one view. At stage k, it attempts all possible joins between partial rewritings of $k-1$ views and partial rewritings of one view.

Query-driven dynamic programming (QDP) is similar to NDP. Nevertheless, to obtain candidate pairs of a k-views rewriting and a one-view rewriting, QDP iterates *over the query nodes*. Let ϕ_x be the embedding of a view v_x to q. When considering query node n, QDP iterates over all view nodes $n_i \in v_i$ such that $\phi_i(n_i) = n$, and such that n_i is annotated ID in v_i. Then QDP iterates over all descendants of n in q, let m be one such descendant, and searches for view v_j and some node $n_j \in v_j$ such that $\phi_j(n_j) = m$ and n_j is annotated with ID in v_j. Thus, QDP does not try to join partial rewritings embedded in disjoint areas of the query, or rewritings with no ID to join on (which NDP attempts to do, and fails).

Query-driven depth-first (QDF) At any point during the search, QDF picks the partial rewriting whose embedding in the query covers the largest number of query nodes, and seeks to combine it with one-view partial rewritings (those covering most query nodes first).

NDP and QDP find a rewriting of k views only after all rewritings of l views, $l < k$. In particular, they find min-size rewritings (Section 4.3.3) before the other rewritings. However, if the smallest rewriting involves many views, say k, NDP and QDP may take a long time building all partial rewritings of sizes $1, 2, \ldots, k-1$. QDF uses the number of query nodes that the partial rewriting embeds to (or covers) as a hint to how many extra views must be joined, to obtain a query rewriting. This may lead QDF to finding a first minimal rewriting fast (if one exists). However, this property cannot be formally

guaranteed in all cases, and indeed one can build counter-examples where QDF's greedy strategy, driven by the coverage of the query nodes, will not lead to finding a rewriting faster than QDP.

4.7.2 Completeness and Termination of TPR algorithm

Theorem 4.7.4 (Completeness of **TPR**). *Algorithm **TPR** is complete, i.e., given a query q and a set of views $\mathcal{V}$ it finds all minimal rewritings of q based on $\mathcal{V}$ that can be obtained through our algebra.*

Proof. The completeness of **TPR** follows readily from the completeness of **TPTransform** (Proposition 4.7.2) and the fact that **TPJoin** (Section 4.7.1.3) tries all possible joins between the *ID*-return nodes of the patterns. □

Theorem 4.7.5 (Termination of **TPR**). *Given a query q and a set of views $\mathcal{V}$, Algorithm **TPR** terminates.*

Proof. (Sketch) The termination of **TPR** follows from the termination of the functions it includes. In particular, we have shown that **TPTransform** terminates in Proposition 4.7.3. **ViewFilter** and **TPJoin** clearly terminate. Finally, **PickPair** also terminates, since there is a finite number of possible candidate pairs of ID-annotated view nodes to join. □

4.8 Rewriting Joined Tree Pattern Queries

We now turn to the problem of rewriting a query in our full language, consisting of several tree patterns (or an XQuery) with value joins, using a set of views $\mathcal{JV}$ of the same kind. Let $jq = \pi_{jq}(\sigma_{jq}(t_1^x \times t_2^x \times \ldots \times t_{n_q}^x))$ be a joined query with $\{t_i^x\}_{1 \leq i \leq n_q}$ being the extended versions (annotating with val all t_i nodes involved in value joins; see Definition 4.3.1) of its tree patterns, and $\mathcal{JV}$ comprise the views $\{v_i\}_{1 \leq i \leq n}$, where each v_i is of the form $\pi_{v_i}(\sigma_{v_i}(t_1^{v_i} \times t_2^{v_i} \times \ldots t_{n_{v_i}}^{v_i}))$. Let $e(v_{j_1}, v_{j_2}, \ldots, v_{j_k})$ be an equivalent algebraic rewriting of jq using the views v_{j_i}, where $1 \leq j_i \leq n$ for all $1 \leq i \leq k$. We have:

Proposition 4.8.1 (Joined-views rewriting). *For each tree pattern t_i of jq, let t_i^x be its extended version, annotating with val all t_i nodes involved in value joins. For every t_i^x, there exists a (LDQT) rewriting, based only on the (extended versions of the) tree patterns of the views $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$ involved in the rewriting e of jq.*

Proof. As stated in Section 4.3.2, every joined tree pattern including the tree patterns t_1, t_2, $\ldots$, t_m, can be written as an algebraic expression of the form $\pi_{jt}(\sigma_{jt}(t_1^x \times t_2^x \times \ldots \times t_m^x))$, where t_i^x, $1 \leq i \leq m$ is the extended version of t_i (see Definition 4.3.1), π_{jt} is a projection retaining all the attributes of the original (non-extended) tree patterns, and $\sigma_{jt} = \sigma_{\bigwedge_{i=1,\ldots,k} vj_i}$ is the conjunction of all the equality predicates that define the value joins. Hence, the joined query jq can be written as:

$$jq \quad = \quad \pi_{jq}(\sigma_{jq}(t_1^x \times t_2^x \times \ldots \times t_{n_q}^x)) \tag{4.4}$$

Algorithm 7: Joined tree pattern rewriting (**JTPR**)

Input : View set $\mathcal{JV} = \{v_1, v_2, \ldots, v_n\}$, query jq
Output: All minimal algebraic rewritings of jq based on $\mathcal{JV}$

$\mathcal{V} \leftarrow \cup_{v \in \mathcal{JV}} (\cup_{t \in v} v.t^x)$ //$v.t^x \leftarrow extend(v.t)$
$L_{rw} \leftarrow \emptyset$, $L_{rw_t} \leftarrow \emptyset$
for $t \in jq$ **do**
  $L_{rw_t}(t) \leftarrow \mathbf{TPR}(\mathcal{V}, t)$
  if $L_{rw_t}(t) = \emptyset$ **then** exit
for $rw = (rw_{t_1} \times rw_{t_2} \times \ldots \times rw_{t_{nq}})$, *such that* $rw_{t_i} \in L_{rw_t}(jq.t_i), 1 \leq i \leq n_q$ **do**
  $discardRW \leftarrow$ false
  for $v.t_i^x$ *appearing* n *times in* rw **do**
    for $v.t_j^x \in v$, $v.t_i^x \neq v.t_j^x$ **do**
      if $v.t_j^x$ *appears* m *times in* rw *and* $m \neq n$ **then** $discardRW \leftarrow$ true
  if !$discardRW$ *and* $vjCheck(rw, jq)$ **then**
    $canonical(rw)$
    replace $v.t_1^x \times \ldots \times v.t_k^x$ in rw with v
    $rw \leftarrow \pi_{vj}(\sigma_{predvj}(rw)))$
    $joinExpression(rw)$
    add rw to L_{rw}
  else discard rw
return L_{rw}

by adding, in the initial projection, the projections that were brought by the extended versions of the tree patterns.

We will show in a constructive way how we can transform the equivalent rewriting expression $e(v_{j_1}, v_{j_2}, \ldots, v_{j_k})$ so as to find a rewriting for each of the tree patterns of jq. Hereafter, for simplicity, we do not take into account the projections. Thus, e can be written as an algebraic expression of the form:

$$jq = \sigma_{jq}(t_1^x \times t_2^x \times \ldots \times t_{n_q}^x) \tag{4.5}$$

and e can be written as:

$$e = \sigma_{vj \wedge ID}(v_{j_1} \times v_{j_2} \times \ldots \times v_{j_k}) \tag{4.6}$$

where σ_{vj} is the conjunction of the predicates corresponding to the value joins that are requested in jq but do not appear in the joined views $v_{j_1}, v_{j_2}, \ldots v_{j_k}$, and σ_{ID} is the conjunction of predicates that correspond to the structural joins between the joined views. Then, we can replace each joined view with the equivalent algebraic expression (without again taking into account the projections) and transform equation (4.6) to the following

one:

$$\begin{aligned} e \;=\; & \sigma_{vj \wedge ID}((\sigma_{vj_1}(v_{j_1.t_1^x} \times v_{j_1.t_2^x} \times \ldots \times v_{j_1.t_{n1}^x}))\times \\ & (\sigma_{vj_2}(v_{j_2.t_1^x} \times v_{j_2.t_2^x} \times \ldots \times v_{j_2.t_{n2}^x}))\times \\ & \ldots \times \\ & (\sigma_{vj_k}(v_{j_k.t_1^x} \times v_{j_k.t_2^x} \times \ldots \times v_{j_k.t_{nk}^x}))) \end{aligned} \tag{4.7}$$

where each joined view v_{j_i}, $1 \leq i \leq k$, is replaced by the expression $\sigma_{vj_i}(v_{j_i.t_1^x} \times v_{j_i.t_2^x} \times \ldots \times v_{j_i.t_{ni}^x})$ and n_i denotes the number of tree patterns participating in the joined view vj_i. By pulling up the selections we get the following expression for e:

$$\begin{aligned} e \;=\; & \sigma_{vj \wedge vj_1 \wedge vj_2 \wedge \ldots \wedge vj_k}(\sigma_{ID} \\ & (v_{j_1.t_1^x} \times \ldots \times v_{j_1.t_{n1}^x} \times v_{j_2.t_1^x} \times \ldots \times v_{j_2.t_{n2}^x} \times \\ & \ldots \times v_{j_k.t_1^x} \times \ldots \times v_{j_k.t_{nk}^x}) \end{aligned} \tag{4.8}$$

As we have already mentioned, σ_{ID} captures all the structural joins that have to be performed between the tree patterns of the joined views that are used in the rewriting. If we push these selections as lower in e as possible, we can create *partitions* of the tree patterns that are structurally connected together. After this transformation, e takes the following form:

$$\begin{aligned} e \;=\; & \sigma_{vj \wedge vj_1 \wedge vj_2 \wedge \ldots \wedge vj_k} \\ & (\sigma_{ID_1}(s_1.t_1 \times \ldots \times s_1.t_{m1}) \times \\ & \sigma_{ID_2}(s_2.t_1 \times \ldots \times s_2.t_{m2}) \times \\ & \ldots \times \\ & \sigma_{ID_l}(s_l.t_1 \times \ldots \times s_l.t_{ml})) \end{aligned} \tag{4.9}$$

where l is the number of the structurally connected partitions of tree patterns, containing m_i ($1 \leq i \leq l$) tree patterns each, σ_{ID_i} denotes the conjunctions of structural predicates that create the i-th partition, while $s_i.t_j$ ($1 \leq j \leq m_l$) is the j-th tree pattern of the i-th partition.

The conjunction of predicates $\sigma_{vj \wedge vj_1 \wedge vj_2 \wedge \ldots \wedge vj_k}$ includes all the predicates corresponding to the value joins that participate in the joined views and to the additional ones that were added during the rewriting process. As e should be (by definition) equal to jq, the value equality predicates that stand on top of both expressions should be equal. Thus, the rest of the expressions should be also equal, which brings us to the following equality (using equations (4.5) and (4.9)):

$$\begin{aligned} t_1^x \times t_2^x \times \ldots \times t_{nq}^x \;=\; & \sigma_{ID_1}(s_1.t_1 \times \ldots \times s_1.t_{m1}) \times \\ & \sigma_{ID_2}(s_2.t_1 \times \ldots \times s_2.t_{m2}) \times \\ & \ldots \times \\ & \sigma_{ID_l}(s_l.t_1 \times \ldots \times s_l.t_{ml}) \end{aligned} \tag{4.10}$$

The left side of the equation (4.10) is a cartesian product over n_q tree patterns, whereas the right side is a cartesian product over l algebraic expressions. We will first prove that $n_q = l$, that is, both sides contain the same number of terms in their cartesian products. From the semantics of the tree patterns, we know that a tree pattern can be embedded to exactly one document at a time. The same holds for each of the algebraic expressions of the right side, as they are built only with structural joins over tree patterns (a structural join

has to be evaluated on the same document). Assume now that $n_q \neq l$ and, in particular, that $n_q > l$. We choose l documents, such that each tree pattern $t_i^x, 1 \leq i \leq l$ from the left side of the equation embeds to exactly one of them. At the same time, each of the l documents contributes to exactly one of the l expressions of the right side. Obviously, we can choose some different documents to contribute to the remaining $n_q - l$ tree patterns of the left side. At such a case, the two parts of the equation will be unequal (contradiction). Hence, $n_q = l$.

We will now prove that each of the tree patterns in the left side of the equation (4.10) is equal to exactly one term of the right cartesian product. It is known in the set theory that for the non-empty sets A, B, C, D the following holds: $A \times B = C \times D \Leftrightarrow A = C, B = D$. The proof is obvious if one considers one-element sets for which $A \neq C$ and forms the cartesian product of them (by contradiction). The same formula holds if we consider the cartesian products of sets of tuples (which derive from the evaluation of some tree patterns or algebraic expressions matching exactly one document at a time). Moreover, the above formula can be readily generalized for cartesian products over more than two terms.

From the above observations, we result in the following equalities:

$$\begin{array}{rcl} t_1^x & = & \sigma_{ID_1}(s_1.t_1 \times \ldots \times s_1.t_{m1}) \\ t_2^x & = & \sigma_{ID_1}(s_2.t_1 \times \ldots \times s_2.t_{m2}) \\ & \vdots & \\ t_{nq}^x & = & \sigma_{ID_l}(s_l.t_1 \times \ldots \times s_l.t_{ml}) \end{array} \tag{4.11}$$

which show that each tree pattern of the query can be rewritten using only the tree patterns of the joined views. □

Proposition 4.8.1 states that a rewriting for the joined query "encapsulates" rewritings for each individual query tree pattern. It is important to notice that *rewriting tree patterns with tree patterns only combines view data "vertically", i.e. by equality or structural joins, whereas value joins connect data "horizontally", across potentially different documents*.

This leads us to Algorithm 8 (called **JTPR**) for rewriting value-joined tree pattern queries. Its basic steps can be summarized as follows: (i) extend all tree patterns in jq and $\mathcal{JV}$, (ii) call Algorithm **TPR** to rewrite each extended query tree pattern with all the extended tree patterns from the $\mathcal{JV}$ views, (iii) for each combination of rewritings output by **TPR**, check if it corresponds (or it can be brought via more selections/projections) to a rewriting of jq, and if yes, output that rewriting. Algorithm **JTPR** uses a simple bucket-like [LRO96] strategy for enumerating combinations of individual tree pattern rewritings. A more efficient strategy as in [PL00] could also be applied. Section 4.2 has presented an example.

A more detailed description of the algorithm is now given. First, we create the set $\mathcal{V}$ of tree pattern views, by extracting from $\mathcal{JV}$ all the tree patterns contained in the joined views (line 1). These tree patterns are extended (just like we did at the end of Section 4.3.2, when specifying their semantics) with the val annotations corresponding to the tree pattern nodes involved in value joins. We then rewrite every tree pattern of jq by invoking Algorithm **TPR** (lines 3-4). If Algorithm **TPR** does not find any result for at least

one tree pattern of jq, the algorithm fails, as it is impossible to find an algebraic rewriting for the whole query, if one of its components cannot be rewritten with the available views.

We now have a list L_{rw_t} containing, for each tree pattern of jq, its set of algebraic rewritings. Then, by choosing one rewriting expression at a time from each set of L_{rw_t}, we create the cartesian product (rw) of these expressions (line 6) and we verify whether it can be used to reach a rewriting of jq based on $\mathcal{JV}$ (lines 8-17). A first observation that leads us to an early pruning step (lines 8-10) is that all tree patterns of a specific joined view should be present in rw, and not only a subset of them. Thus, the number of appearances of the tree patterns of the same query in rw should be the same; otherwise we discard the current expression. A second pruning step is performed by invoking the function $vjCheck$ (line 11). A rw passes successfully through this step if the value joins of the joined views of the tree patterns that participate in it are less restrictive than the ones imposed by the query. At the same time, we compute the additional selections and projections that should be imposed in our rewriting expression by virtue of the possible additional value joins that appear in jq.

At this point, it has been verified that our current cartesian product can result in a rewriting of jq and the corresponding rewriting expression should now be created (lines 12-16). In line 12, we transform rw to a *canonical form* that will enable us to apply further transformations on it. The canonical form of rw is defined as a projection over a selection over a cartesian product of the tree patterns that participate in rw. We observe now that the current canonical rewriting expression contains the tree patterns of the joined views and not the actual joined views we have in our disposal. Hence, we replace the tree patterns with the corresponding joined views to get a canonical expression where only existing views participate (line 13). To this expression we apply, if needed, an additional projection and/or a selection (line 14), which refer to the additional value joins of the query and were computed during the $vjCheck$. Finally, for efficiency reasons, we seek to reduce the number of cartesian products in the rewriting, by replacing as many of them as possible with joins, which are likely to have more efficient physical implementations (line 15).

4.9 Related Works

Several works have addressed XML query rewriting using views. *Maximally contained rewriting* is studied in [LWZ06]. *Equivalent rewriting using a single view* is considered in [MS05, XO05, YLHA03]. The works directly comparable to ours study *equivalent XML query rewriting using multiple materialized views* [ABMP07, BOB$^+$04, CDO08, TYÖ$^+$08, ODPC06]. Unlike our work, [ODPC06] does not allow views to store IDs, which limits view join possibilities. Node IDs appear in the views in [BOB$^+$04, CDO08, TYÖ$^+$08, ABMP07]. The algorithms in [ABMP07, CDO08] do not require any special ID property, whereas [ABMP07] exploits also structural IDs if available. The rewriting algorithm in [BOB$^+$04] requires that views store, next to each node ID, the complete label path from the document root to the node. Similarly, [TYÖ$^+$08] requires an expressive class of IDs [LLCC05], encapsulating the labels (and IDs) of all the ancestors of the node. However, such expressive IDs are not avail-

able in all cases. Our algorithm can work with any type of ID, and exploits (but does not require) structural IDs when available. From this viewpoint, it most directly compares with [CDO08], which shows that the rewriting problem for the XPath$^{\{/,//,[]\}}$ dialect (with single return nodes) is coNP-hard, and identifies restricted settings for which the problem is polynomial. The algorithm in [CDO08] does not exploit structural IDs, and does not guarantee minimality of the rewriting expressions. We note that the algorithms in [BOB$^+$04, TYÖ$^+$08] do not provide completeness guarantees.

Concerning the view language, XPath 1.0 dialects in which a view may only store *ID and/or content for one node* are used in [BOB$^+$04, CDO08, LWZ06, MS05, TYÖ$^+$08, XO05, YLHA03], while [CC10, PZIÖ06] assume *only IDs are stored, and from all view nodes*. In contrast, views in [ABMP07] and in this work store IDs and/or values and/or contents for an arbitrary subset of view nodes, leading to more flexibility, but also making rewriting more complex. The rewriting algorithm of [ABMP07] requires structural knowledge about the database under the form of a Dataguide, which is not needed in this work.

Works complementary to ours [CC10, PZIÖ06] describe efficient physical storage models for XML views, and efficient holistic twig join algorithms for joining views on IDs. Our focus here is on identifying logical rewriting plans, on which the physical optimization techniques of [CC10, PZIÖ06] could also apply.

Closely related problems studied in XML databases are materialized view selection [TYT$^+$09], query containment [MS04, TH04] and minimization [AYCLS02].

Early elements of this work were informally presented in [MZ09], whereas our algorithms were demonstrated in [KZ10].

4.10 Conclusion and Future Work

A powerful tool to speed up query evaluation, in particular within an XML query evaluation context, is to use materialized views, which store partial pre-computed query results. Such views can bring important performance gains but require a view-based query rewriting step preceding execution. In this Chapter, we have considered the problem of finding equivalent rewritings for queries expressed in an XQuery dialect corresponding to rich tree pattern languages connected through value joins. We have designed the algorithm **TPR** which builds rewritings for single-tree-pattern queries in a bottom-up fashion, by searching first for all possible one-view rewritings, and then building larger rewritings in a bottom-up fashion, adding views one by one to a rewriting. By design, **TPR** only develops minimal rewritings, i.e., such that no view can be removed while preserving the rewriting semantics. Based on **TPR**, we have devised the algorithm **JTPR** which rewrites joined tree pattern queries using similar joined tree pattern views. We have shown that **TPR** and **JTPR** are sound and complete for the respective languages they consider.

Throughout the **TPR** we exploit the duality between tree patterns, on one side, and algebraic expressions, on the other. The simplest partial rewriting we consider simply scans a view, thus its algebraic expression is naturally equivalent to the view's tree pattern. At the end of the rewriting process though, the algebraic rewriting plan may be quite complex, yet it has to be equivalent to the query tree patterns, in order to be an equivalent

rewriting of that query. Through **TPR**, thus, we juggle with algebraic operators (to build ever larger rewritings) and equivalent tree patterns (to identify the tree pattern to which the rewriting is equivalent, when such a tree pattern exists). We have introduced two main procedures: one for attempting to *adapt*, through algebraic transformations, a partial rewriting by adding on it constraints from a query tree pattern (**TPTransform**), and the other for *combining* through a join, two partial rewritings into a larger one (**TPJoin**). We formalized the semantics of these procedures, and shown that when called from the **TPR** algorithm, they enable the latter to explore all equivalent minimal rewritings built on the views with the help of the algebra - in other words, to be complete. The completeness of algorithm **JPTR** follows from that of **TPR**.

The algorithm **TPR** we have described follows the overall structure of the one we described in [MKVZ11], however, we have completely re-worked and formalized it from the bottom up. In this Chapter, we also defined basic properties on joined tree patterns, and provided efficient criteria for establishing these procedures.

Many continuations of this work are possible. First, we may extend it to nested patterns, representing several nested XQuery FLWR blocks [ABM05]. There are also numerous possibilities to improve its performance. Finally, we are considering the automated recommendation of materialized XML views (on which work has already started in our team [KMV12]) based on a lattice (AND-OR graph) capturing the way views can be rewritten based on each other. This is likely to be more efficient than invoking the rewriting algorithm; at the same time, the AND-OR graph for XML queries and views is probably quite complex, thus efficient methods for reasoning without completely building it are needed.

Chapter 5

Distributed Sharing of Annotated Documents

Apart from their multiple benefits when used in a centralized setting, materialized views can also be of great importance in a distributed setting, as explained in Chapter 2. In such a setting, where the data transfers constitute the predominant cost factor in query execution, the intelligent placement of data across the network can expedite query evaluation considerably.

This is the rationale behind ViP2P (standing for *V*iews in *P*eer-to-*P*eer), a DHT-based distributed platform developed in our group (see `http://vip2p.saclay.inria.fr`), exploiting materialized views for the efficient dissemination of XML data among peers. The rewriting algorithm presented in Chapter 4 is part of the query engine of ViP2P in order to optimize query evaluation. The architecture of ViP2P was the main topic of a previous PhD thesis [Zou09] and appeared in [KKMZ12]. Details on the platform, along with an extended more recent experimental evaluation, are presented in [KKMZ11].

In this Chapter, we present AnnoVIP, a system built on top of ViP2P for the sharing of documents with annotations. Taking the idea of ViP2P a step further, we consider large corpora of structured documents (such as HTML and XML Web pages) and semantic annotations (typically expressed in RDF), which further complement these documents. Documents and annotations may be authored independently by different users or programs. In this distributed scenario, AnnoVIP allows efficiently disseminating such content in a DHT-based P2P networks, by taking advantage of materialized views.

This Chapter is largely based on the demonstration [KZ10, CRKM$^+$10]. It is worth noting that AnnoVIP motivated the problem of rewriting tree pattern queries *with value joins*, which was the focus of the previous Chapter.

5.1 Motivation and outline

In recent years, more and more software tools, including the most user-friendly ones, such as text editors, have started to export their contents into some *structured document format*, such as HTML or XML. Moreover, *annotations* have become very popular as a

means to add information to a given document. HTML Meta tags, Dublin Core [Dup] and social networks' tagging are among the most common methods to express annotations. Here, we designate by annotation any simple statement in the style of the RDF standard [www04b], attaching to a given subject (or resource, such as a document, or a small portion of text) a named property, with a certain value (see Chapter 2 for more details on the XML and RDF data models).

Using documents *and* annotations provides the flexibility to handle a variety of application scenarios in which documents or RDF alone would not be suitable. As an example, consider a Web page containing a news item, annotated by a human reader or a text analysis tool to point out the person names appearing in the page, her positions within various institutions, etc., or to express subjective opinions regarding the document. One could suggest modifying the Web page to incorporate the additional information of the annotations. However, this is not always feasible, since the author of the annotations may be distinct from, and have no control over, the author of the original document; furthermore, the original document should be readable also to those that are not interested in the extra information. Using only RDF to model all the content, on the other hand, is not appropriate since end users are familiar with, and expect to use, structured documents.

Documents and annotations are at the center of the WebContent [web] project, in which our group was involved. The project is focused on building and maintaining warehouses of enhanced Web documents on specific topics, e.g., market survey for the EADS european company (with offices in several countries) or an intelligence survey/warehouse concerning news from online media, bloggers, etc., concerning a specific area of the world.

Content publication in WebContent applications is inherently distributed. Documents coming from the Web are fetched by crawlers running at different sites, possibly reformatted, translated from one language to another, and published by the respective sites. Similarly, documents and annotations authored by domain experts are published from their sites. WebContent applications require that all sites be able to exploit all the published contents. One could have considered uploading all published content to a single site. However, this raises scalability issues, which may require acquiring dedicated hardware, and introduces a single point of failure.

The ViP2P project In 2008-2011 (thus partially before the beginning of this thesis), our group has worked on developing the ViP2P platform (the ViP2P project website is at `http://vip2p.saclay.inria.fr`). ViP2P is a symmetrical, peer-to-peer platform, based on a distributed hash table [DZD+03] (or DHT, in short). We opted for a DHT-based architecture since it guarantees upper bounds on the number of hops needed in order to route a given message in the peer network. At the core of content sharing in ViP2P stand *materialized views over the whole network content*. Each peer may locally store some XML content, and may also define views, describing patterns of interconnected documents and annotations, that the peer is interested in. These views are stored in a local repository at the peer. Once a view is established, its definition will be indexed in the DHT network. When documents are published, by looking up in the DHT, the publishing peer learns if its new content may contribute to some view, and if so, it sends the respective data to the view. After publication, this lookup is repeated periodically to

identify contributions to views defined later on. Thus, views are updated over time, in the manner of long-running, de-centralized subscriptions.

A further step in content sharing in ViP2P is *materialized view-based query rewriting*. Here, we consider the situation when a peer issues an ad-hoc query, which it has not declared as a local view. The peer then looks up in the DHT the existing view definitions, and may rewrite its query based on the views. This process produces a logical algebraic rewriting plan, which is distributed, since it involves views potentially scattered across the DHT network. Subsequently, ViP2P's optimizer and distributed execution engine take over in computing the distributed plan's results.

ViP2P's query and view language as of mid-2009 had two important limitations:

1. *value joins* were not supported, thus, each view or query was a single tree pattern;
2. *parent edges* were not supported, thus, each pattern edge was $//$.

While quite severe for the expressive power, these limitations lead to a greatly simplified query rewriting algorithm [Zou09].

Extending ViP2P into AnnoViP Motivated by distributed annotated content applications such as those of the WebContent project outlined above, we have developed AnnoVIP by extending ViP2P (i) to address the above limitations and (ii) to enable the modeling and querying of annotations at very fine granularity, i.e., that of a word in a text paragraph. While initially developed for AnnoVIP both extensions are now part of ViP2P itself; the expanded rewriting algorithm, capable of handling joined tree pattern queries, is Algorithm **JTPR** described in Section 4.8. Moreover, from a user perspective, AnnoVIP extends ViP2P by the support for RDF data next to XML.

Beyond the complex rewriting algorithm which we presented in Chapter 4, the contributions of AnnoVIP are:

1. we demonstrate how, based on distributed joined tree pattern views and queries, one can annotate distributed content in a peer-to-peer setting, and subscribe to other peers' annotations;
2. we accordingly extend the ViP2P architecture to materialize views (or, equivalently, update subscriptions) expressed by *joined* tree patterns. The extension is not straightforward since a view may need to join data appearing in the future, anywhere in the DHT network.

In the sequel, Section 5.2 provides an overview of ViP2P. Then, Section 5.3 discusses content publishing in AnnoVIP, whereas Section 5.4 describes querying. We relate AnnoVIP to existing works in Section 5.5, and then we conclude.

5.2 Overview of ViP2P: Efficient XML Management in DHT Networks

The ViP2P platform has been developed starting in 2008 in the Gemo (then Leo, now Oak) group at Inria Saclay. It focuses on the large-scale management of distributed XML data in a structured *peer-to-peer* (P2P) setting, namely a DHT network. To provide users

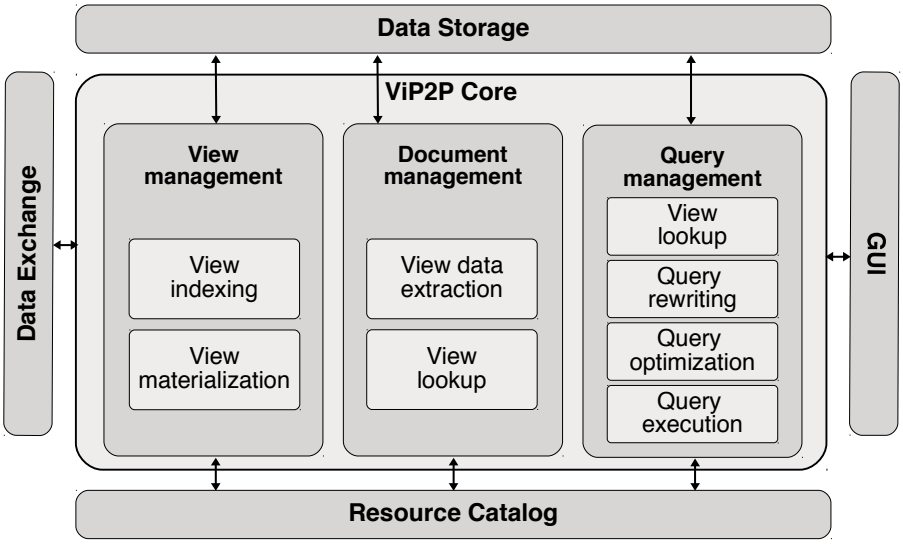


Figure 5.1: Architecture of a ViP2P peer.

with *precise and complete* answers to their requests for information, it is assumed that the requests are formulated by means of a structured query language, and the system must return complete results. That is, if somewhere in the distributed peer network, an answer to a given query exists, the system will find it and include it in the query result.

XML data flows in ViP2P can be summarized as follows. XML documents are *published* independently and autonomously by any peer. Peers can also formulate *subscriptions*, or long-running queries, potentially matching documents published before, or after the subscriptions. The results of each subscription query are *stored* at the peer defining the subscription, and the definition of it is *indexed* in the peer network. Finally, peers can ask *ad-hoc queries*, which are answered in a snapshot fashion (based on the data available in the network so far) by exploiting the existing subscriptions, which can be seen as *materialized views*.

We will now briefly analyze the architecture of ViP2P (depicted in Figure 5.1), together with its basic functionalities. We first describe the *auxiliary modules*.

Resource catalog uses the FreePastry DHT [Fre, RD01] to provide the underlying DHT layer used to keep peers connected, and to index and look up views.

Data exchange module is responsible *for all data transfers of significant size* and relies on Java RMI. Note that the DHT is not used for such transfers, because it has been shown that it becomes the bottleneck when sending important volumes of data [AMP+08].

Data storage Within each peer, view tuples are efficiently stored into a native store, built using the BerkeleyDB [BDB].

GUI enables users to publish views, documents and pose queries.

We now move to the description of the *core modules*.

Document management determines to which views the peer's documents may contribute

data by using the Resource catalog (**View definition lookup** module), and extracts and sends this data to the appropriate consumers (**View data extraction** module).

View management consists of two sub-modules:

View indexing Based on some strategies, it indexes the available view definitions, by adding entries to the Resource catalog.

View materialization This module is responsible for keeping views up-to-date with the latest data arriving at the network.

Query management comprises the following sub-modules for query evaluation:

View lookup This module, given a query, performs a lookup in the Resource catalog to retrieve the view definitions that may be used to rewrite the query.

Query rewriting Given a query and a set of view definitions, this module exploits the view-based query rewriting algorithm, presented in Chapter 4, to produce a logical plan, which, when evaluated based on the views, produces exactly the results required by the query.

Query optimization This module receives a logical plan that is output by the Query rewriting module, and translates it to an optimized physical plan. The optimization concerns both the logical (join reordering, push selections/projections, etc.) and physical (dictating the exact flow of data during query execution) level.

Query execution This module provides a set of physical operators, implementing the standard iterator-based execution model [Gra90], which can be executed by any ViP2P peer in a distributed manner.

5.3 Content publication in AnnoVIP

Having provided a brief overview of the ViP2P architecture in the previous Section, we now detail the publication of documents, annotations and views in AnnoVIP. Section 5.3.1 describes the content which one may publish, whereas Section 5.3.2 outlines the publication process. To illustrate our explanation, a simple AnnoVIP instance over six peers is depicted in Figure 5.2. Next to each peer, the Figure shows its published XML documents (such as xml_1, xml_2, etc.), annotations (denoted rdf_1, rdf_2, etc.) and/or materialized views (v_1, v_2, etc.).

5.3.1 Content model

The first kind of content we consider consists of XML documents. Each document d published by peer p has a URI allowing to uniquely determine d inside p and in the whole network. For example, Figure 5.3 shows an article on the financial crisis, published by user Alice. This could be the $xml_{article}$ document published by peer p_4 in Figure 5.2.

Annotations can target content at very different granularity levels. Thus, one can annotate a document, an element, a text node, or even a fragment of text, e.g., a phrase of particular significance, or a person's name appearing in some text. Therefore, we

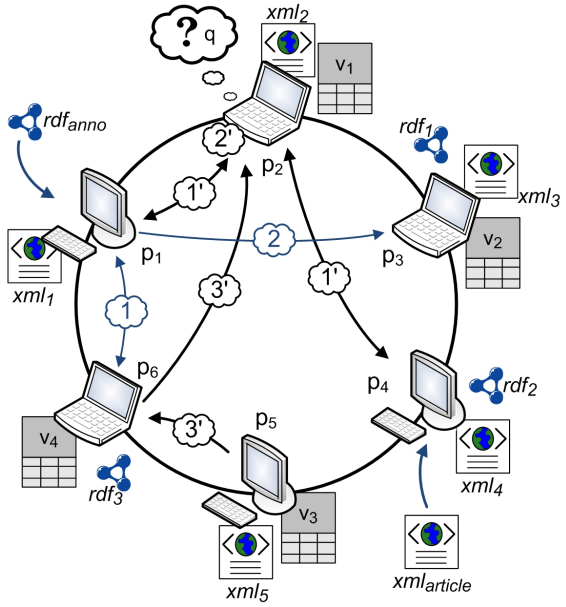


Figure 5.2: AnnoVIP overview.

```
<article>
  <publishInfo>
    <author>Alice</author>
    <year>2012</year>
    <country>...</country>
  </publishInfo>
  <headline>Financial Crisis</headline>
  <topic>economy</topic> <body>...</body>
</article>
```

Figure 5.3: Sample published document $xml_{article}$.

consider that any fragment of a document d, whatever its size, has a URI. Such URIs are implemented by (offset, length) pairs identifying the fragment in the serialization of d. Moreover, the URI of d can be easily obtained from the URI of any fragment of d. This holds in many common URI schemes, such as XPointer [www01], where $d.URI$ is a prefix of all the URIs of elements in d.

Our model assumes that any XML element has a child labeled *URI*, whose value is the actual URI of the element. However, URI-labeled nodes are virtual, that is, they do not actually appear in the elements (although as we will explain, they are needed for querying documents and annotations). In a similar manner, any fragment can be seen as a node, endowed with a URI. Without loss of generality, as well as for conciseness, we focus only on elements hereafter.

The second class of content we consider concerns annotations, which may be either produced by human users, possibly with the help of some tools, or by automated modules (e.g., recognizing named entities within a document). While several dialects can be

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999-rdf-syntax-ns#"
         xmlns:anno="http://gemo.inria.fr/annotate/">
  <rdf:Description rdf:about="http://gemo.inria.fr/article.xml#body">
    <anno:authName>Bob</anno:authName>
    <anno:authCountry>France</anno:authCountry>
    <anno:date>26-6-2009</anno:date>
    <anno:rating>interesting</anno:rating>
    <anno:seeAlso>"http://gemo.inria.fr/article2.xml"</anno:seeAlso>
  </rdf:Description>
</rdf:RDF>
```

Figure 5.4: Sample annotation rdf_{anno}.

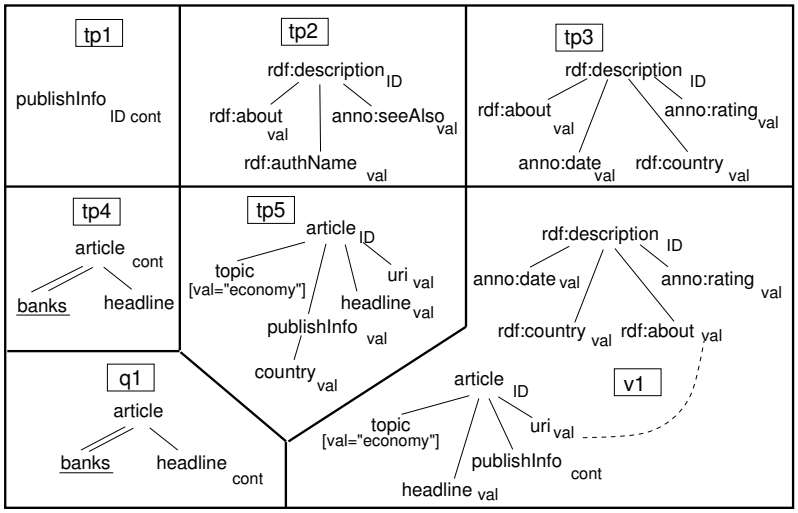


Figure 5.5: Sample queries and views.

used for annotations, such as Dublin Core [Dup], HTML Meta tags, or microformats, for simplicity, we consider that all annotations have been brought to an RDF format. Thus, the basic unit of content here is a triple $t{=}(s, p, o)$, specifying the value o of property p for the resource s. We assume triples are serialized following the XML syntax for RDF [www04c]. As customary in RDF, s, p and o range over the set of all URIs, plus the set of literal (string) values in the case of o.

For instance, let us consider how user Bob at peer p_1 in Figure 5.2 produces the new annotation rdf_{anno}. Assume Bob discovers the $xml_{article}$ published by Alice at p_4 and decides to annotate the *body* of the article as being "interesting". Bob also suggests another article (*article2.xml*), which he finds related. The corresponding RDF annotation appears in Figure 5.4, together with the author's (i.e., Bob's) profile.

5.3.2 Peer-to-peer views in AnnoVIP

Views and queries are defined in (a slightly extended version of) the joined tree pattern language described in detail in Chapter 4 of this thesis. Figure 5.5 shows some examples. The extension we bring for the needs of our annotation scenarios consists of allowing *word nodes* in our tree patterns. A word node can only appear as a leaf tree pattern node; in the Figure, a node corresponding to the word w is denoted $\underline{\text{w}}$. Tree pattern semantics (Section 4.3.2.1) extends easily to incorporate such word nodes, by considering that a text word is a child of its closest enclosing element or ancestor node. *Due to the special role we attach to URIs, we impose that a URI-labeled view node always appears as a child (not descendant) of another node in the view.*

In Figure 5.5, pattern tp_1 stores the structural ID and the content of all *publishInfo* elements. Pattern tp_2 stores the author's name, as well as the suggested documents for all annotations, while tp_3 keeps the date, country and rating of annotations. Pattern tp_4 stores the content of all articles containing the word "banks" and having a headline. Finally, tp_5 stores the URI, the headline and the country of the publisher of all articles about the economy. Observe that although element URIs are not stored in the original document, they must be actually stored in a view, such as tp_5. More complex views can be obtained by joining tree patterns based on some value equality predicates, such as view v_1. Observe that the presence of value joins in the language is crucial for capturing the connections between content and annotations on that content, via equality predicates of the form rdf:$about.val$=$uri.val$.

View materialization We now consider how published views are filled in with data. Assume peer p creates a view v. Then, when a peer p_d publishes a document d affecting v, p_d needs to find out that v exists. To that effect, view definitions are *indexed for document-driven lookup* as follows. For any label (node name or word) appearing in the definition of the views $v_1, v_2, \ldots, v_k$, the DHT (in particular, the Resource catalog module, described in Section 5.2) will contain a pair where the key is the label, and the value is the set of view URLs $v_1, v_2, \ldots, v_k$.

When a peer p_d publishes a document d, p_d performs a lookup with all d labels (node names or words) to find a superset S_a of the views that d might affect. Then, p_d evaluates $v(d)$ for each $v \in S_a$. If v consists of a single tree pattern, the results of $v(d)$ are sent directly to the peer holding $v(d)$. For example, Figure 5.2 shows the events taking place when user Bob publishes rdf_{anno} at p_1. Firstly, he performs a look up to determine (a possible superset of) the view definitions to which the new annotation may add some tuples (step 1). In our Figure, p_6 holds a view definition index entry referring to such a view. Upon receiving the view definition, say that of v_2 which p_3 holds, p_1 extracts the tuples corresponding to $v_2(rdf_{anno})$ and sends them to p_3 (step 2), which appends them to the view extent. Observe that the contribution of rdf_{anno} to the view could be evaluated locally at p_1, assuming the view is a tree pattern.

Assume now that a view in S_a is a joined tree pattern view, i.e., it contains more than one tree patterns and some value joins between them. It is not possible to materialize such a view only by considering document d, as p_d needs to know if and where, in the whole network, some other content may satisfy a value join with d. A first solution could be to

maintain, instead of the joined pattern view, one view per each tree pattern, and compute view tuples incrementally as new tuples are added to each tree pattern, in the style of incremental maintenance for join views [GMS93]. However, this may lead to accumulating a significant amount of (useless) data, if, e.g., many documents matching one view tree pattern are published, which do not join with any other document or annotation.

We will now describe a more efficient technique for the case when join predicates *do not* involve URI attributes. Assume, as above, peer p_d publishes a document d. Let $v' \in S_a$ be a joined pattern view, consisting of two tree patterns, tp_1 and tp_2, with a value join between nodes n_1 of tp_1 and n_2 of tp_2. If there is an embedding of tp_1 in d, instead of storing the whole tuple in tp_1, we store only the value of n_1 and the URI of d. Assume now a document d' is published and there is an embedding of tp_2 in d'. The publishing peer searches the (value, URI) pairs and finds immediately the documents with which d' could be joined, by comparing its n_2 value with the available n_1 values. Thus, the peer receives the appropriate documents, joins them and provides new tuples to the view extent.

Moreover, we have devised a particular technique to treat the materialization of views including joins over URI attributes. This is the case whenever a view queries documents with annotations, as the view will involve joins over virtual URI attributes. Notice that annotations are necessarily published *after* the content they refer to. Thus, when a new document is published, it will not contribute (yet) to join views requiring specific annotations over the document. On the contrary, when a newly published annotation matches a join view, the URI of the annotated element appears in the annotation and the document enclosing this element can thus be identified. The peer that has published the annotation then asks the document peer to compute its corresponding tuples, which are joined with the tuples extracted from the annotation and sent for storage at the site of the view.

5.4 Query rewriting using views

We now consider the processing of a query, such as q in Figure 5.2, posed at a peer p_2, which has not declared q as a local materialized view. First, p needs to find out which views in the network could possibly be used to answer q. This relies on the same view definition indexing used for view maintenance. Then, p runs a view-based rewriting algorithm to find complete rewritings of q using the existing materialized views. Finally, one rewriting is picked by the optimizer and evaluated over the distributed peers (see Section 5.2 for the modules used in the querying process).

The rewriting of the queries based on the views was the subject of Chapter 4. Here, we present the query evaluation process by example.

In Figure 5.2, query q is posed by user Carole at peer p_2. To find view definitions relevant to q, p_2 performs a DHT lookup (step 1′). Assume that peers p_1 and p_4 return relevant view definitions. Peer p_2 then tries to rewrite q based on these views (step 2′). The outcome of the algorithm is a logical algebraic plan based on some views, in our example v_3 and v_4, which are stored in peers p_5 and p_6 respectively. Subsequently, p_2 transforms the rewriting to a distributed physical plan, which is executed in a distributed fashion in the network (step 3′). For instance, v_3 is sent from p_5 to p_6, where it is joined

with the local v_4 and the result is sent back to p_2.

Query q_1 asks for the headlines of articles mentioning banks. A possible rewriting could just use tp_4 to navigate through the content of element *article* and then project the headline. Now consider a more complex query q_2, requiring the headlines of all articles published in 2009 that were annotated as interesting today in France. Query q_2 can be rewritten using tp_3 and tp_5 in the obvious way, if these patterns are available as materialized views. The drawback is that this rewriting still requires evaluating one (value) join. However, if view v_1 were available, q_2 could be answered more efficiently, without evaluating any costly joins.

5.5 Related works

AnnoVIP relates to many works on XML indexing in DHT networks [AMP+08, GWJD03, RM09], over which it improves by allowing to declare and exploit complex materialized views to speed-up the processing of specific application queries. A recent, more general survey on P2P data management can be found in [Abe11].

As already mentioned, we have implemented AnnoVIP on top of VIP2P. The system is developed in Java, using the Pastry DHT and BerkeleyDB for storing materialized views. VIP2P's scalability has been established in large-scale experiments involving up to 1000 peers in a country-wide WAN [Gri], thousands of documents and hundreds of views.

Within the WebContent project [web], another DHT-based platform was previously developed, integrating two types of DHT content indexes [AAC+08]. However, this still did not provide sufficient leeway to establish efficient data access support structures. Moreover, the biggest performance problems of that system [AAC+08] were due to the frequent joins generated by document-and-annotations queries.

Works related to the efficient evaluation of RDF queries are discussed in Chapter 2 (Section 2.2.3).

5.6 Conclusion and Future Work

Annotated documents are used in a variety of applications nowadays, many of which are inherently distributed. Users of such systems may pose queries referring both to documents and annotations, and requiring information residing anywhere in the whole network. Therefore, devising techniques to efficiently manage annotated documents in such scenarios is important. In this Chapter, we presented AnnoVIP, a distributed system that is based on a DHT overlay network. In AnnoVIP, users can express their interests as views, which are materialized in the system. Subsequently, when other users pose queries, these views are exploited by our view-based query rewriting algorithm in order to improve query evaluation. AnnoVIP was built on top of ViP2P, a system that has been deployed on up to 1000 peers and whose scalability has been thoroughly tested [KKMZ11].

More generally, we witness an important proliferation of technical means to issue and exchange annotations, in particular when exchanging opinions in social or collaborative

sites, or when processing documents automatically, by text, language and semantic analysis. This justifies the interest in models and tools enabling to capture both documents and annotations, if possible without converting one to the format of the other. An effort in this area has recently started in the group [GKK+11a, GKK+11b], and continues within a separate PhD thesis. Some more details about this work are given in Chapter 6 (Section 6.2).

Chapter 6

Conclusion and Perspectives

Web data is becoming available at extreme rates nowadays, with big part of it being expressed in XML (tree-structured) and RDF (graph-structured). Efficiently manipulating Web data is undoubtedly crucial, but at the same time challenging, due to the size and the complex structure of the available data. In this thesis we have focused on the management of XML and RDF data based on materialized views in order to expedite the query evaluation over such data.

6.1 Thesis Summary

We have focused on two basic problems, which we summarize below.

RDF view selection We have considered the context of a Semantic Web database in which both explicit (present in the original database) and implicit (derived by applying the entailment rules of an RDF Schema) data is present. In this context:

- We tackled the problem of selecting a set of views to be materialized in the database in order to minimize a combined cost of query evaluation, view storage and update maintenance. We modeled our problem as a state optimization problem, inspired from an existing relational approach.
- Starting from an initial state (which corresponds to materializing exactly the query workload), we devised various strategies and heuristics to search for the most appropriate set of views.
- We investigated how we can efficiently take into account implicit data in the view selection process, based either on database saturation or query reformulation. A new query reformulation algorithm had to be created for our query and view language. Moreover, we proposed an original method (named post-reformulation) to incorporate query reformulation in the view selection, without increasing the number of queries upon which the search is done (and thus the complexity of the search)
- We implemented all our algorithms and experimentally showed the benefits of our approach. Most importantly, our search algorithms can scale up to hundreds of queries, achieving considerable cost benefits even at such workloads.

XML view-based query rewriting We then tackled the problem of answering XQuery

queries using XQuery views. In particular:

- We considered an expressive fragment of XQuery, which brings significant extensions to the dialects considered in previous rewriting works (while lacking support for star nodes that some of these works considered). Our query and view language is represented by joined patterns, consisting of several tree patterns. Each tree pattern can have multiple return nodes and multiple return attributes for each node. Moreover, value joins are supported between the nodes of tree patterns.
- Given this query and view language, we devised a novel sound and complete rewriting algorithm, which finds complete (no need to access the data) and minimal (no view can be removed and still have a rewriting) rewritings.
- We established various properties for the joined tree patterns we consider, including containment, equivalence and minimality, and devised practical algorithms to efficiently check for such properties.
- We presented ways to transform a tree pattern to another one by means of an algebra.
- We exploited our rewriting algorithm in a distributed setting for the efficient sharing of XML documents, complemented with RDF annotations.

6.2 Perspectives

The management of RDF data is a highly active area of research. Although numerous works have been presented the last years to this respect, there are still many opportunities for improvement in creating a store capable of efficiently accommodating every possible dataset and query workload. Starting from the work of this thesis on the view-based management of Web data, we outline below various avenues for future work.

RDF view selection Our work on RDF view selection can be extended to support a more expressive language, including aggregate queries, which are important for decision support systems and data warehouses. As far as the efficiency of the search is concerned, new search strategies applying additional heuristics to guide the search could be devised. At the same time, partitioning the workload in clusters that share common characteristics would possibly lead to finding states with even lower cost and parallelization opportunities could be exploited. As for the implicit triples, it would be interesting to find a solution that lies between query reformulation and database saturation combining the best of both worlds.

XML view-based rewriting Our rewriting algorithm could be extended to support *-labeled nodes in the patterns, as well as to support nesting and optionality. Most importantly though, further techniques for optimizing the algorithm should be exploited, given the high complexity of the problem. Among them, cost-based heuristics could be employed in order to avoid joining views that are likely to lead to inefficient rewritings. Moreover, at the moment we are investigating ways to answer a given query based on views that may rely on other views.

Self-organizing Web data management systems Exploiting materialized views for

query optimization purposes is one way of tuning a database system based on knowledge for the query workload. Taking the idea of our work a step further, one can consider an XML or RDF data management system whose design is automatically adapted according to the characteristics of the given dataset and by exploiting the possible knowledge for the kind of queries that will be posed. Adaptivity can be considered in three main areas:

- *Adaptive storage and indexing.* The way an XML/RDF dataset is stored and indexed plays a crucial role on the performance of query evaluation. Unfortunately though, there is no single storage and indexing scheme that works best on all types of data and queries. Hence, it is important to automatically select the most suitable way for storing and indexing the dataset by analyzing the specificities of the given dataset and, if available, of the expected query workload.
- *Partitioning.* One can observe that parts of the same database may have very different characteristics. In fact, some queries may access parts of the data in an OLAP fashion, others may be purely transactional, whereas others may combine both features. Moreover, some parts of the data are expected to be updated at much higher rates than others. Based on these observations, one can partition the data according to their features and store each partition in a different way.
- *Dynamic adaptivity.* In the above two cases, data and queries are considered to be known in advance and to remain fixed. However, in many systems, updates in the data and changes in the queries occur. This may alter the characteristics of the dataset or queries, causing the performance even of a well-adapted system to degrade over time. Therefore, it is of interest to devise ways to automatically adapt the system in the presence of such updates.

Joint manipulation of XML and RDF data The idea of jointly manipulating XML and RDF data started while we were building the AnnoVIP system (see Chapter 5 of this thesis) for the sharing of XML documents with RDF annotations. However, AnnoVIP was actually operating over pure XML data, after transforming the RDF annotations to an XML/RDF format. Transforming data from one data model to another, means that we are no longer capable of exploiting techniques that have been developed for the original data model in which the data was expressed. To this end, we are currently working on XR [GKK+11a, GKK+11b], a hybrid model between XML and RDF, following and combining the W3C's XML, URI and RDF standards so as to enable semantically rich annotations and querying. XR makes any XML node a first-class citizen by assigning it a URI and connecting it to a graph of RDF triples. We have devised a query language and are now investigating ways to build an execution engine that can enable both model-specific, as well as cross-model optimizations. Work on XR continues within a separate PhD thesis.

Publish/Subscribe view-based system over Fast Data This is an ongoing work, which started during my internship at the University of California at San Diego, focusing on building a distributed view-based publish/subscribe system. The system consists of a database server and multiple application servers, and is capable of accommodating a significant number of clients. The clients subscribe to the data sources of the database server that interest them using views, and the aim of the system is to keep the clients' views

up to date with the least possible latency, under some resource constraints (e.g., processing power, storage capacity, network bandwidth). To this end, multi-query optimization, partitioning and distribution techniques are exploited. Finalizing the corresponding algorithms and architecture is part of my future work.

Bibliography

[AAC+08] Serge Abiteboul, Tristan Allard, Philippe Chatalic, Georges Gardarin, A. Ghitescu, François Goasdoué, Ioana Manolescu, Benjamin Nguyen, M. Ouazara, A. Somani, Nicolas Travers, Gabriel Vasile, and Spyros Zoupanos. Webcontent: efficient P2P warehousing of web data (demo). *PVLDB*, 1(2), 2008.

[Abe11] Karl Aberer. Peer-to-peer data management. *Synthesis Lectures on Data Management, Morgan & Claypool Publishers*, Volume 3:p.87–94, 2011.

[ABM05] Andrei Arion, Véronique Benzaken, and Ioana Manolescu. XAMs: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web*. Morgan Kauffman, 1999.

[ACP12] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629–638, 2012.

[AD98] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

[AGR07] P. Adjiman, F. Goasdoué, and M.-C. Rousset. SomeRDFS in the semantic web. *Journal on Data Semantics*, 8, 2007.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[AMP+08] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[AMR+12] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2012.

[ASC+09] Parag Agrawal, Adam Silberstein, Brian F. Cooper, Utkarsh Srivastava, and Raghu Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *SIGMOD Conference*, pages 179–192, 2009.

[AYCLS02] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[Bas] BaseX. The XML database. http://basex.org/.

[BC06] Angela Bonifati and Alfredo Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.

[BCH+06] Kevin S. Beyer, Roberta Cochrane, M. Hvizdos, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Matthias Nicola, Fatma Özcan, Hamid Pirahesh, Normen Seemann, Ashutosh Singh, Tuong C. Truong, Robbert C. Van der Linden, Brian Vickery, Chun Zhang, and Guogen Zhang. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006.

[BDB] Oracle Berkeley DB Java Edition. http://www.oracle.com/technetwork/products/berkeleydb.

[BFK05] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005.

[BGKM12] Francesca Bugiotti, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. RDF Data Management in the Amazon Cloud. In *Workshop on Data analytics in the Cloud (DanaC 2012)*, 2012.

[BGMS11] Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. Algebraic incremental maintenance of XML views. In *EDBT*, pages 177–188, 2011.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006.

[BK08] Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.

[BK09] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4), 2009.

[BKS02] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[BOB+04] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[CC10] Ding Chen and Chee-Yong Chan. ViewJoin: Efficient view-based evaluation of tree pattern queries. In *ICDE*, pages 816–827, 2010.

[CDES05] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, pages 1216–1227, 2005.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, , and M. Tommasi. Tree automata techniques and applications, 2007.

[CDO08] Bogdan Cautis, Alin Deutsch, and Nicola Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[CGL+96] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. Algorithms for deferred view maintenance. In *SIGMOD Conference*, pages 469–480, 1996.

[CGL+07] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *JAR*, 39(3), 2007.

[CHS02] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *VLDB J.*, 11(3):216–237, 2002.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[CL10] Roger Castillo and Ulf Leser. Selecting materialized views for RDF data. In *ICWE Workshops*, 2010.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[CRCM12] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Building Large XML Stores in the Amazon Cloud. In *DMC - Data Management in the Cloud Workshop - 2012*, 2012.

[CRKM+10] Jesús Camacho-Rodríguez, Konstantinos Karanasos, Ioana Manolescu, Alin Tilea, and Spyros Zoupanos. Viewing a world of annotations through AnnoVIP. In *Bases de Données Avancées*, 2010.

[CV93] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *real* conjunctive queries. In *PODS*, 1993.

[DCDK11] Vicky Dritsou, Panos Constantopoulos, Antonios Deligiannakis, and Yannis Kotidis. Optimizing query shortcuts in RDF databases. In *ESWC*, 2011.

[DFS99] Alin Deutsch, Mary F. Fernández, and Dan Suciu. Storing semistructured data with STORED. In *SIGMOD Conference*, pages 431–442, 1999.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04. USENIX Association, 2004.

[DGL00] Oliver M. Duschka, Michael R. Genesereth, and Alon Y. Levy. Recursive query plans for data integration. *J. Log. Program.*, 43(1):49–73, 2000.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD Conference*, pages 145–156, 2011.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, pages 459–470, 1999.

[Dup] Dublic Core metadata initiative. http://www.dublincore.org/.

[DZD+03] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured P2P overlays. In *IPTPS*, 2003.

[eXi] Open source native XML database. Exist: Open Source Native XML Database.

[FM00] Thorsten Fiebig and Guido Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB (Selected Papers)*, pages 125–136, 2000.

[Fre] Freepastry, an open-source implementation of Pastry. http://freepastry.org/FreePastry/.

[GJÖY09] Torsten Grust, H. V. Jagadish, Fatma Özcan, and Cong Yu. XQuery processors. In *Encyclopedia of Database Systems*. Springer US, 2009.

[GKK+11a] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents. In *First International Workshop on Searching and Integrating New Web Data Sources*, 2011.

[GKK+11b] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents. In *Journées de Bases de Données Avancées*, Rabat, Morocco, October 2011.

[GKLM10a] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: a storage tuning wizard for RDF applications. In *CIKM*, 2010.

[GKLM10b] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. Materialized View-Based Processing of RDF Queries. In *Bases de Données Avancées*, 2010.

[GKLM11] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. RDFViewS: A Storage Tuning Wizard for RDF Applications. In *Bases de Données Avancées*, 2011.

[GKLM12] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View selection in Semantic Web databases. *PVLDB*, 5(1), 2012.

[GL01] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[GM05] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD Conference*, 1990.

[Gri] Grid'5000 network infrastructure. https://www.grid5000.fr/.

[Gup97] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.

[GWJD03] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[IHW01] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating network-bound XML data. *IEEE Data Eng. Bull.*, 24(2):20–26, 2001.

[JKV06] T. S. Jayram, Phokion G. Kolaitis, and Erik Vee. The containment problem for real conjunctive queries with inequalities. In *PODS*, pages 80–89, 2006.

[JLVV01] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, and Panos Vassiliadis. *Fundamentals of Data Warehouses*. Springer, 2001.

[KCS11] Shahan Khatchadourian, Mariano P. Consens, and Jérôme Siméon. Having a ChuQL at XML on the Cloud. In *AMW*, 2011.

[KKMZ11] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. The ViP2P Platform: XML Views in P2P. Technical Report RR-7812, INRIA, November 2011.

[KKMZ12] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. ViP2P: Efficient XML management in DHT networks. In *ICWE*, 2012.

[KMV12] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.

[KV00] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.

[KZ10] Konstantinos Karanasos and Spyros Zoupanos. Viewing a world of annotations through AnnoVIP. In *ICDE*, 2010.

[LDK+11] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. Rewriting queries on SPARQL views. In *WWW*, 2011.

[LIK06] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating conjunctive triple pattern queries over large structured overlay networks. In *ISWC*, 2006.

[LKDL12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for SPARQL. In *ICDE*, 2012.

[LLCC05] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In *VLDB*, 2005.

[LM09] Zhen Hua Liu and Ravi Murthy. A decade of xml data management: An industrial experience report from oracle. In *ICDE*, pages 1351–1362, 2009.

[LP08] Kostas Lillis and Evaggelia Pitoura. Cooperative XPath caching. In *SIGMOD*, 2008.

[LPPS12] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. Static analysis and optimization of semantic web queries. In *PODS*, 2012.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[LWZ06] Laks V. S. Lakshmanan, Hui Wang, and Zheng (Jessica) Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[MASS08] Angela Maduko, Kemafor Anyanwu, Amit P. Sheth, and Paul Schliekelman. Graph summaries for subgraph frequency estimation. In *ESWC*, 2008.

[MKK08] Iris Miliaraki, Zoi Kaoudi, and Manolis Koubarakis. XML Data Dissemination Using Automata on Top of Structured Overlay Networks. In *WWW*, 2008.

[MKVZ11] Ioana Manolescu, Konstantinos Karanasos, Vasilis Vassalos, and Spyros Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[MS04] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1), 2004.

[MS05] Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In *VLDB*, pages 315–326, 1999.

[MZ09] Ioana Manolescu and Spyros Zoupanos. XML materialized views in P2P. DataX workshop (not in the proceedings), 2009.

[NDM+01] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Kumar Gupta, and Rushan Chen. The Niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1), 2008.

[NW09] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[NW10a] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[NW10b] Thomas Neumann and Gerhard Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1), 2010.

[ODPC06] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[PCS+04] Shankar Pal, Istvan Cseri, Gideon Schaller, Oliver Seeliger, Leo Giakoumakis, and Vasili Vasili Zolotov. Indexing XML data stored in a relational database. In *VLDB*, pages 1134–1145, 2004.

[PH01] Rachel Pottinger and Alon Y. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3):182–198, 2001.

[PL00] Rachel Pottinger and Alon Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[PZIÖ06] Derek Phillips, Ning Zhang, Ihab F. Ilyas, and M. Tamer Özsu. InterJoin: Exploiting indexes and materialized views in XPath evaluation. In *SSDBM*, pages 13–22, 2006.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ICDSP*, November 2001.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[RM09] Praveen Rao and Bongki Moon. An internet-scale service for publishing and locating XML documents (demo). In *ICDE*, 2009.

[Ros11] Riccardo Rosati. On the finite controllability of conjunctive query answering in databases under open-world assumption. *J. Comput. Syst. Sci.*, 77(3):572–594, 2011.

[SBCL00] Kenneth Salem, Kevin S. Beyer, Roberta Cochrane, and Bruce G. Lindsay. How to roll a join: Asynchronous incremental view maintenance. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 129–140. ACM, 2000.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[SGK+08] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.

[Sof] OpenLink Software. Virtuoso universal system. http://virtuoso.openlinksw.com/.

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

[SSB+08] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[STP+05] Arsany Sawires, Jun'ichi Tatemura, Oliver Po, Divyakant Agrawal, and K. Selçuk Candan. Incremental maintenance of path expression views. In *SIGMOD Conference*, pages 443–454, 2005.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[tCM07] Balder ten Cate and Maarten Marx. Navigational XPath: calculus and algebra. *SIGMOD Record*, 36(2):19–26, 2007.

[TH04] Igor Tatarinov and Alon Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD*, 2004.

[TLS01] Dimitri Theodoratos, Spyros Ligoudistianos, and Timos K. Sellis. View selection for designing the global data warehouse. *Data Knowl. Eng.*, 39(3), 2001.

[TS97] Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD Conference*, 2002.

[TYÖ+08] Nan Tang, Jeffrey Xu Yu, M. Tamer Özsu, Byron Choi, and Kam-Fai Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[TYT+09] Nan Tang, Jeffrey Xu Yu, Hao Tang, M. Tamer Özsu, and Peter A. Boncz. Materialized view selection in XML databases. In *DASFAA*, 2009.

[UPS07] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. In *AAAI*, 2007.

[W3C04] W3C. Extensible markup language (XML) 1.0, 2004.

[W3C07a] W3C. XML path language (XPath) 2.0, January 2007.

[W3C07b] W3C. XQuery 1.0: An XML query language, January 2007.

[W3C07c] W3C. XQuery 1.0 and XPath 2.0 data model (XDM), January 2007.

[w3c07d] XPath Functions and Operators. www.w3.org/TR/xpath-functions, 2007.

[W3C08] W3C. Extensible markup language (XML) 1.0, November 2008.

[web] WebContent, the Semantic Web platform (RNTL project). www.webcontent.fr.

[WKB08] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *PVLDB*, 1(1), 2008.

[WSKR03] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[www] The Barton data set. http://simile.mit.edu/wiki/Dataset:_Barton.

[www01] XML Pointer Language. http://www.w3.org/TR/WD-xptr, 2001.

[www04a] RDF. Available at www.w3.org/RDF/, 2004.

[www04b] RDF: Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[www04c] RDF/XML Syntax Specification. http://www.w3.org/TR/rdf-syntax-grammar/, 2004.

[www04d] RDF Vocabulary Description Language 1.0: RDF Schema. Available at www.w3.org/TR/rdf-schema/, 2004.

[www08] SPARQL query language for RDF. Available at www.w3.org/TR/rdf-sparql-query/, 2008.

[www11] Experimental results. Available at http://rdfvs.saclay.inria.fr/experiments, 2011.

[XML] XML Schema. http://www.w3.org/TR/XML/Schema.

[XO05] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[YLHA03] Liang Huai Yang, Mong-Li Lee, Wynne Hsu, and Sumit Acharya. Mining frequent query patterns from XML queries. In *DASFAA*, 2003.

[ZLFL07] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD Conference*, pages 533–544, 2007.

[ZMC+11] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gStore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

[Zou09] Spyros Zoupanos. *Efficient peer-to-peer data management*. PhD thesis, Inria Saclay and Université Paris Sud, December 2009.