UNIVERSITY OF CALIFORNIA, SAN DIEGO

Uncovering the Full Potential of Data Services

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Nicola Dan Onose

Committee in charge:

Alin Deutsch, Chair
Richard K. Belew
Michael J. Carey
Bertram Ludäscher
Yannis Papakonstantinou
Jérôme Siméon
Victor Vianu

2009

The dissertation of Nicola Dan Onose is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2009

DEDICATION

To my mother, Maria Onose, to whom I owe everything I am.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

During my PhD years, my advisor, mentors and co-workers helped me to define and reach work goals, develop my research skills and shape my career. I am sincerely grateful to all of them for contributing to my success.

My first and most earnest acknowledgement goes to my advisor, Alin Deutsch, who from the beginning trusted and encouraged me on the difficult road of finding a dissertation topic, forming a doctoral committee and carry out the thesis proposal. Alin's role was vital for improving my research, writing and presentation skills. Our long research meetings and long feedback sessions while preparing a talk were the most instructive part of my graduate school education. Alin was also the person on which I could always rely for receiving career advice and planning life after graduation.

My co-authors and collaborators at UCSD, Yannis Papakonstantinou, Emiran Curtmola and Yu Xu, together with Alin Deutsch, made an important contribution to the material that formed the basis of Chapter 2 in this dissertation. Their help in designing the algorithm, implementing the experiments and the online demonstration and writing the SIGMOD 2006 paper, was essential for the success of the project on rewriting XML queries using XML query views.

I would like to give a special thanks to Bogdan Cautis, with whom I wrote the papers that laid the foundation for Chapters 3–5. Bogdan provided an invaluable aid in many ways, from solving complicated technical problems to giving feedback to practice talks before conferences.

I would also like to thank my internship mentors and paper co-authors from industrial research labs, who helped me broaden my research and gain experience in an industrial environment. Jérôme Siméon and Mary Fernández helped me publish my first research papers and were a great source, along the years, of fun projects and ideas for experimenting with new types of applications. While an intern at IBM, I also had the chance to work with Kristoffer Rose, whose expertise in programming languages proved to be a great asset when delving into the intricacies of XML query and update languages.

To Michael Carey and Vinayak Borkar I owe a great internship at BEA,

resulting in the implementation of a new feature for a commercial product. Later, we published an industrial paper describing our work that I presented at VLDB 2007, which has been my favorite conference until now.

Sergey Melnik and Philip Bernstein mentored me on a challenging project that needed a practical solution for a problem with non-trivial theoretical implications. This gave me the opportunity to explore research topics going beyond my area of expertise in query optimization and rewriting queries using views. Working on a paper with Phil and Sergey that described our work on that project was an excellent opportunity to improve my writing style under their expert and attentive guidance.

Last, but not least, I would like to thank all the members of my doctoral committee for the advice provided both during the thesis proposal phase, which helped me delimit the scope of the dissertation, and later, when working on this dissertation and on the defense talk.

Chapter 2 is currently being prepared for submission for publication of the material. Onose, Nicola; Deutsch, Alin; Papakonstantinou, Yannis; Curtmola, Emiran; Xu, Yu. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is a revised reprint of the material published in SIGMOD 2006. Onose, Nicola; Deutsch, Alin; Papakonstantinou, Yannis; Curtmola, Emiran. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in WebDB 2008. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 4 is currently being prepared for submission for publication of the material as it may appear in Theory of Computing Systems, 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 4, in part, is a reprint of the material as it appears in ICDT 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 5, in part, is a reprint of the material as it will appear in VLDB 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola; Vassalos, Vasilis.

VITA

2003	Engineering Degree in Computer Science, Politehnica University, Bucharest
2004	Engineering Degree from Ecole Polytechnique, Paris, and ENSIMAG, Grenoble
2005-2009	Graduate Research Assistant, University of California, San Diego
2007	Master of Science in Computer Science, University of California, San Diego
2009	Doctor of Philosophy in Computer Science, University of California, San Diego

PUBLICATIONS

Nicola Onose, Jérôme Siméon, "XQuery at Your Web Service", *WWW2004*.

Irini Fundulaki, Arnaud Sahuguet, Guillaume Giraud, Nicola Onose, Nicolas Pombourcq, Daniel Lieuwen, "Share your data, keep your secrets", *SIGMOD*, 2004.

Mary Fernández, Nicola Onose, Jérôme Siméon, "Yoo-Hoo! Building a Presence Service with XQuery and WSDL", *SIGMOD*, 2004.

Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, Emiran Curtmola, "Rewriting Nested XML Queries Using Nested Views", *SIGMOD*, 2006.

Vinayak R.Borkar, Michael J. Carey, Dmitry Lychagin, Till Westmann, Daniel Engovatov, Nicola Onose, "Query Processing in the AquaLogic Data Services Platform", *VLDB*, 2006.

Mary Fernández, Trevor Jim, Kristi Morton, Nicola Onose, Jérôme Siméon, "Highly Distributed XQuery with DXQ", *SIGMOD*, 2007.

Mary Fernández, Trevor Jim, Kristi Morton, Nicola Onose, Jérôme Siméon, "DXQ: A Distributed XQuery Scripting Language", *XIME-P*, 2007.

Nicola Onose, Vinayak R.Borkar, Michael J. Carey, "Inverse Functions in the AquaLogic Data Services Platform", *VLDB*, 2007.

Giorgio Ghelli, Nicola Onose, Kristoffer Rose, Jérôme Siméon, "A Better Semantics for XQuery with Side-Effects", *DBPL*, 2007.

Giorgio Ghelli, Nicola Onose, Kristoffer Rose, Jérôme Siméon, "XML Query Optimization in the Presence of Side Effects", *SIGMOD*, 2008.

Nicola Onose, Rania Khalaf, Kristoffer Rose, Jérôme Siméon, "A Restful Workflow Implementation on Top of Distributed XQuery", *XIME-P*, 2008.

Bogdan Cautis, Alin Deutsch, Nicola Onose, "XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency", *WebDB*, 2008.

Bogdan Cautis, Alin Deutsch, Nicola Onose, "Querying Data Sources That Export Infinite Sets of Views", *ICDT*, 2009.

Bogdan Cautis, Alin Deutsch, Nicola Onose, Vasilis Vassalos "Efficient Rewriting of XPath Queries Using Query Set Specifications", *VLDB*, 2009.

ABSTRACT OF THE DISSERTATION

Uncovering the Full Potential of Data Services

by

Nicola Dan Onose
Doctor of Philosophy in Computer Science
University of California San Diego, 2009
Alin Deutsch, Chair

Making use of available services when building Web applications is a major challenge for today's developers. I address this challenge by using a declarative interface for data-centric Web services (aka data services), which are published as queries over a source schema. Programmers simply write queries over the source schema and rely on the system to automatically translate them to calls to existing data services. Thus, programmers can focus on extracting the data they need, without having to understand the definition or the implementation of each individual service.

This dissertation discusses the main underlying technical problem, that of deciding whether a query can be translated into service calls. We consider two settings: when the system cannot do any post-processing and hence can issue only one service call (I call that expressibility) and when it is able to issue several calls and combine the results (I call it support). Expressibility and support are studied both for services that are listed individually and for compactly represented services (using grammar-like or Datalog formalisms).

I also present contributions to extending the underlying service infrastructure with new features, several of which were added to the Distributed XQuery (DXQ) framework. DXQ is an XML query and scripting language with support for side effects, distribution, parallelism, which I also used as implementation platform for workflow languages.

Chapter 1

Data Services

ABSTRACT OF THE CHAPTER

Currently deployed (data-centric) Web Services publish only their input and output types, without describing the data manipulation they perform. This makes it hard for developers to find and compose services to answer their requests. I propose adding to the service specification a description of their behavior in terms of data manipulations performed on a public view of the source schema. I call these services *data services* and I propose algorithms for automatically answering user requests by one or a combination of data services.

1.1 Problem statement

Due to the arrival of the Web and the increasing number of programs built from heterogeneous packages, the IT world is moving towards an architecture based on services. The purpose of many of these services is to act as wrappers over data sources that usually expose restricted query interfaces, because of performance, business model or security reasons. For instance, a service allowing to track Fedex packages asks for the tracking number of the package, instead of allowing unlimited access to the status of all packages.

I refer to these services as *data services* and in my research I addressed fundamental problems related to their large scale usage. The large number of

available Web services and resources prevents developers from having a global overview required to manually select and use the data services necessary for their tasks. Moreover, Web data sources (from online communities to ultra large scale systems) become more and more complex, and need to publish very large sets of services. In this context, it is necessary to have an automated procedure for deciding if a client request can be answered by the source. This procedure is implemented by a module called *adapter*, which may reside either on the client or on the server. If the request needs to be answered without post-processing, the adapter checks if the user request is equivalent to one of the published services and I call it *expressibility* [CDO09] (see Figure 1.1). If post-processing can be done, then the adapter checks if the request is equivalent to a combination of published services, as in Figure 1.2, and I call the corresponding decision problem *support* [CDO09].

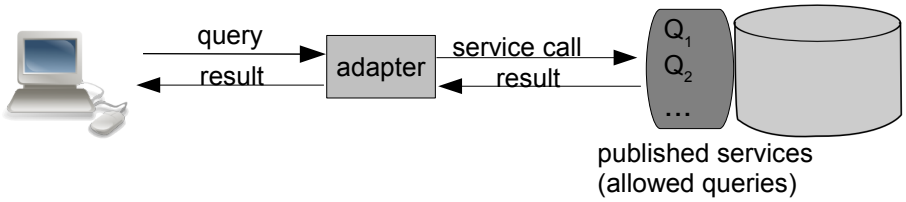


Figure 1.1: Expressibility: answer a query using one service call

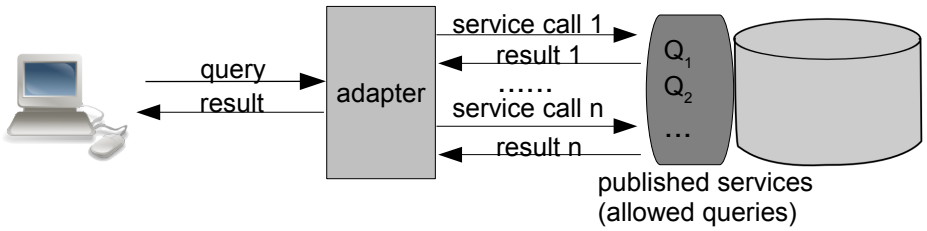


Figure 1.2: Support: rewrite a query using service calls

Currently deployed Web Services publish only the input and output type of the service, without describing the data manipulation performed. This makes it hard or even impossible for the service publisher to specify what the purpose of the service is at declaration/registration time or for a service registry to discover services relevant for a given user query. Therefore, a lot of the data manipulation logic in present systems is coded manually, based on additional expertise and knowledge. This may lead to errors and it becomes unfeasible once the set of published services becomes very large. The solution I propose is in the spirit of classical database technologies–whose success was based on presenting to the user a declarative interface (the query language), hiding the lower level implementation– and provides automatic checking procedures for expressibility and support.

1.2 Proposed solution: declarative data services

The solution I propose is to adopt *declarative data services*, a type of data services that specify their behavior in terms of data manipulation statements. I have been working on extending their framework in order to respond to the ever increasing demands of data-centric Web applications.

They generalize the concept of parameterized views [DLN07] from classical database research, as their behavior is defined by queries and data update statements with parameters bound at call time to arguments provided by the caller. A declarative data service may access any data source (database, Web service, files) for which there is an appropriate interface. I distinguish between declarative data services, built essentially upon data management technologies, and the Semantic Markup for Web Services [OWL], which is a Semantic Web framework.

Since declarative data services are known in literature as *data services*, from now on I will refer to them simply as data services.

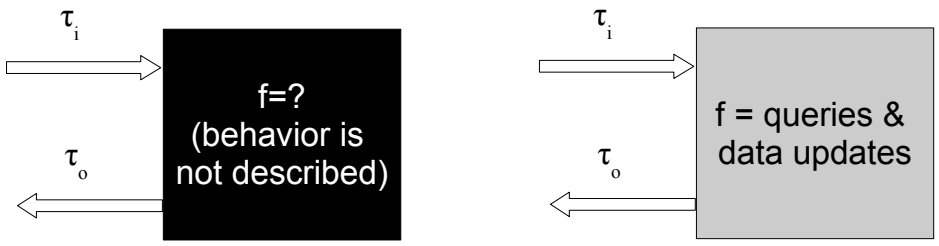
1.2.1 Service declaration

In general, any service, in order to be usable, needs to be able to interact and make itself accessible to other services and applications. How to declare services

is thus a key issue.

A service can be in general characterized in terms of its input/output signature $\sigma = (\tau_i : \tau_o)$ and its behavior [BKM07], specified as a function with signature σ.

In current applications, Web Services are typically described using Web Service Description Language(WSDL) [CCMW], a W3C standard, which only exposes the input/output signature $(\tau_i : \tau_o)$ in terms of XML Schema types. WSDL offers no information about the behavior of the service, which is then made available as a black box, with input type τ_i and output type τ_o, as in Figure 1.3(a). Therefore implied knowledge and and a lot of individual expertise in dealing with a data sources is needed on the developers side, making their task very complex. (For convenience, I will call Web Services based on WSDL, WSDL services.)



(a) Traditional (WSDL) Services (b) Data Services

Figure 1.3: Data services expose service behavior

In contrast, the behavior of a data service is not a black box anymore, but it is specified in terms of data manipulation tasks over a public view of the source schema, as suggested in Figure 1.3(b). This declarative characteristic simplifies several tasks, such as finding and composing the services relevant for building a Web application.

Section 6.1 shows how WSDL can be extended to publish the data manipulations performed by data services. Thus, we can reuse the infrastructure offered by the current standards.

Publishing information related to the schema may seem dangerous to some-

one used to the idea that security should be enforced by unveiling as little information as possible. However, the view that the data owner publishes may be different from the physical input schema of the service, thus sensitive information can be hidden. Traditional WSDL services were already exposing schemas, but only of the service input and output. If we want to have automatic tests for expressibility and support and to enable query optimizations, we also need to publish a schema on which the service data manipulations can be defined. I argue that an intuitive understanding of this schema was already part of the domain expertise that the developers were using in order to understand what data a service was providing.

Since XML is the de facto standard for representing data on the Web, XML query [CD99, BCF+] and update languages [CFM+08, GRS06, CFF+06] represent the main means for specifying data services. This dissertation will focus on data services that perform only queries (no side effects). The extension with data updates is left as future work.

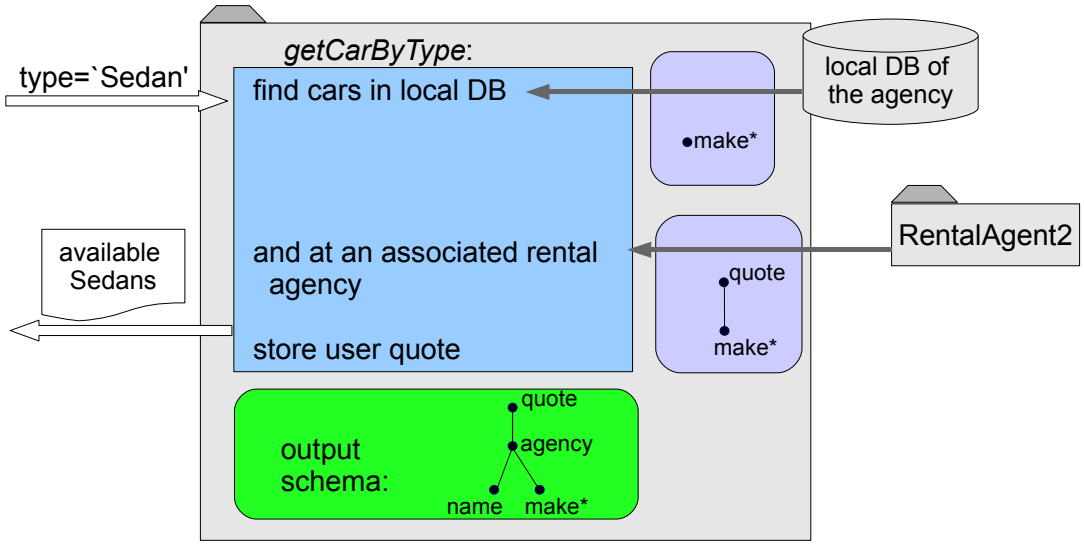


Figure 1.4: Example of a car rental data service

Figure 1.4 contains a high-level description of a data service that returns all cars available at a rental agency that have a given type (Sedan, in our example). There is only one input argument, the type of the car, which is a string. The data

service starts by finding a list of relevant cars in the local database. It also queries the data of an associated rental agency, which groups the cars by quotes. It also stores the user quote, thus performing a local data update. It then returns a result conforming to an output schema in which each quote has an agency subelement, specifying the name of the agent providing the cars and the makes of available cars.

Data services have been used before, both in academic research [PDP03] and in industrial research, in the context of the BEA AquaLogic Data Services Platform (ALDSP) [Car06, BCL+06]. The BEA Data Services have many similarities with ours: they are defined using XQuery [BCF+] queries and use data from local data sources or provided by other services. However, ALDSP does not provide mechanisms for expressibility and support, which are the main topic of this work.

Publishing a large (even infinite) number of services A requirement particular to data services comes from the fact that data sources sometimes want to publish larger classes of parameterized queries than just a few fixed function calls. This is currently done via Web Service API's, targeted towards general purpose programming languages, such as the API's provided by Yahoo or Google for managing one's online data and services. Dealing with detailed API's complicates the application logic and the development process. And it can get even more complex for data sources that integrate other data sources, because they may want to publish even larger sets of queries.

EXAMPLE 1.2.1. *Consider the example of a web portal, such as BarnesAndNoble.com selling books that exposes six attributes: title, author, price range, format, age, subject. Suppose that the data owner is interested in publishing services that receive as input arguments a subset of the attributes and return the corresponding values of another attribute. Such services could be specified by queries such as "find the subject of the book, given the author and the format". Or "find the title of the book, given the age, the subject and the format". We can see that even for such simple services with unary output, there are* $2^6 - 1 = 63$ *possible queries. Based on business or performance goals, the data owner may choose to publish any subset of*

those queries. Hence the set of published services can be very large, exponential in the size of the schema.

For data sources publishing very large number of services, listing the service definitions as individual queries has major drawbacks both for the developers and for the publishers. Developers cannot have an overview of all services and thus it is difficult for them to find the services they need. Publishers need to specify each service individually and then to maintain a large number of service specifications.

In the case when a source publishes an infinite number of services, as in the example below, listing all services becomes even impossible.

EXAMPLE 1.2.2. *A travel agency exports services defined by queries that return trips, of unbounded length, such that Paris is reachable from the last visited point either by train or by bus. Thus, the data source exports an infinite number of services, corresponding to one of the two types of chains depicted in Figure 1.6. Figure 1.7 gives examples of such queries: V_1 looks for a trip covering three touristic spots, such as from the last one there is a train connection to Paris. V_2 is similar, except that it checks the existence of a bus connection to Paris.*

The solution I propose is to publish *compact specifications* of very large or infinite sets of services. I will present such encodings, which are similar in spirit to Datalog [AHV95] programs (which specify a union of conjunctive queries) or to grammars [UH79] (which finitely specify a language).

In the remainder of this section, I discuss the main challenges for solving expressibility and support, both for listed and compactly encoded services. I give detailed explanations and present solutions in the chapters that follows.

1.2.2 Listed Data Services

I consider first the case in which the set of published services is listed as a finite set of queries. The main challenge comes from the fact that we are using XML queries, whose properties have not been fully studied, and which allow complex language constructs.

Expressibility For listed data services, checking expressibility means checking equivalence between the user query and each of the queries from the published set. XQuery [BCF+] is known to be Turing complete, hence there is no decision procedure for expressibility for the entire XQuery language. I will present algorithms for expressibility (and support) that are complete for an important fragment of XQuery, and become sound tests in general.

Support Since all services are explicitly given, support reduces to a classic problem in database research, that of rewriting queries using views (see [Hal01] for a survey). In the following we will call the services published by a data source *views*.

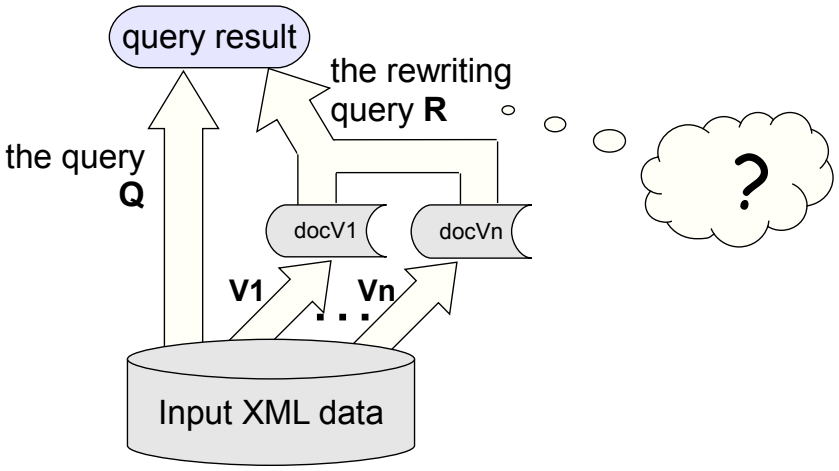


Figure 1.5: Rewriting queries using views

Figure 1.5 gives a high-level description of rewriting queries using views in the context of XML data sources. We are given a query Q, in a language $\mathcal{L}_Q$, over some input XML data, and a set of views defined by queries $V_1, V_2 \ldots V_n$, in a language $\mathcal{L}_V$, over the same data. The results of $V_1, V_2 \ldots V_n$ are available as $doc(V_1), doc(V_2), \ldots doc(V_n)$ (which can be either materialized or not). We are asking whether we can formulate a query R, in a language $\mathcal{L}_R$, over the view documents to obtain the same result as by running Q over the original input. If such a query exists, we say that R is a *rewriting* of Q using the views $V_1, \ldots V_n$.

Discussion Besides the equivalent rewritings discussed above, previous literature also studied rewritings that can approximate the user query, either from below—called maximally contained rewritings [Hal01]—or from above—called minimally containing rewritings [DLN07]. Such approximation methods are outside the scope of this dissertation, as I considered that the natural approach is to lay the theoretical foundations for equivalent rewritings first. Therefore, in the following, any usage of the term *rewriting* will implicitly mean *equivalent rewriting*.

While for expressibility, we only care about the language of the query $\mathcal{L}_Q$ and of the views $\mathcal{L}_V$, for support (aka rewriting queries using views) we also need to take into account the language $\mathcal{L}_R$ in which the rewriting R is expressed. One important primitive that we would like to have in $\mathcal{L}_R$, besides navigation into XML documents, is intersection, because it is one of the basic features of relational query languages and has been traditionally used for rewriting relational queries using views. In the case of XML queries, dealing with intersection is non-trivial, as a non-empty intersection of two paths means that they both navigate to the same node in the XML document, which in many cases may be hard to check statically. I discuss in Chapter 3 the effect of intersection when rewriting several fragments of the XPath [CD99] language.

1.2.3 Compactly Encoded Services

If the set of services is given by a compact encoding, instead of being listed, new challenges are added to solving expressibility and support.

Expressibility for compactly encoded services consists in deciding whether there is a service (defined by a query) in the published set that is equivalent to the user query. The main challenge comes from the fact that one cannot enumerate all services and check equivalence, since the set of services can be very large or even infinite.

Support for compactly encoded services consists in deciding whether there is a finite subset of services included in the (possibly infinite) published set that can

be used to construct a rewriting. The same observations hold as for expressibility: we cannot re-use the algorithms for the finite case (for rewriting queries using views) because we don't have explicitly all the published services/views.

Compactly Encoded Relational Services As a preliminary work on compactly encoded services, I first looked at relational queries and views, because their properties are better understood. Large sets of (conjunctive) queries can be encoded compactly as expansions of Datalog programs, as it was noticed in previous literature [LRU99, VP00].

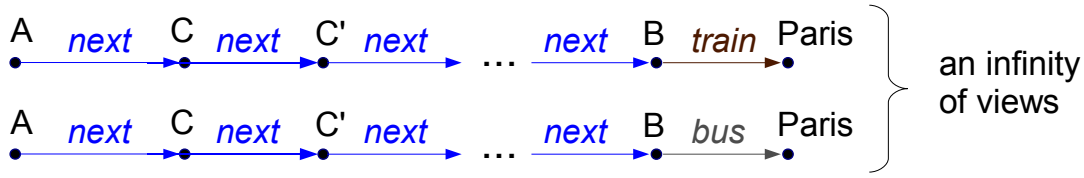


Figure 1.6: Example of an infinite set of services $\mathcal{V}$

For instance, the set of views from Figure 1.6 can be encoded by the following Datalog program $\mathcal{P}$, which is similar to the one presented in Example 4.1.1:

$$
\begin{aligned}
ans(A, B) &:- next(A, C), itin(C, B) \\
itin(C, B) &:- next(C, C'), ind(C', B) \\
itin(C, B) &:- next(C, B), train(B, \text{“}Paris\text{”}) \\
itin(C, B) &:- next(C, B), bus(B, \text{“}Paris\text{”})
\end{aligned}
$$

Consider now a user query asking for trips going through three touristic spots, such as from the last one Paris can be reached by train, bus and shuttle:

$$
\begin{aligned}
q(A, B) :- \; & next(A, C), next(C, B), train(B, \text{“}Paris\text{”}), \\
& bus(B, \text{“}Paris\text{”}), shuttle(B, \text{“}Paris\text{”})
\end{aligned}
$$

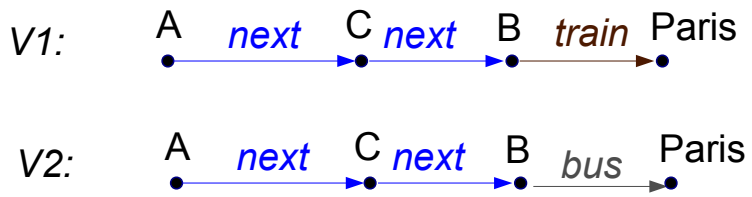


Figure 1.7: Two views from the infinite set $\mathcal{V}$

q cannot be answered using the expansions of $\mathcal{P}$, as the rules of the programs do not mention the shuttle. However, if we add a constraint saying that *any stops connected by train and by bus are also connected by shuttle*, then q can be rewritten as the conjunction of the views V_1 and V_2 from Figure 1.7. This example illustrates how constraints can enable a large class of rewritings that were not possible in their absence.

My work [CDO09] consisted in identifying restrictions under which expressibility and support become decidable in the presence of constraints (embedded dependencies) on the data source, analyzing their complexity and presenting solutions that are provably correct. Moreover, the solution I proposed for support runs in EXPTIME, improving the algorithms from previous research, which reported 2-EXPTIME [LRU99] or NEXPTIME [VP00] upper bounds.

Compactly Encoded XML Services A natural follow-up was then to analyze the problem of answering XML queries using XML data services. This combined the expertise acquired in the case of a finite number of views [ODPC06, CDO08] and the intuition from studying expressibility and support for relational queries [CDO09].

Since there was no previous work on expressibility and support for XML queries, deciding how to encode the services was also an open problem. The final choice was the Query Set Specification Language (QSS) [PDP03], a formalism for

encoding families of XPath queries that is similar to context free grammars.

Consider the example of a tourism agency that allows looking for various kinds of trips. Figure 1.8 gives an example of a QSS for this application that allows searching for museums on tour trips or on guided trips. The trips can be nested, meaning that the trip on a lower nesting level is a secondary trip of the trip on the higher level.

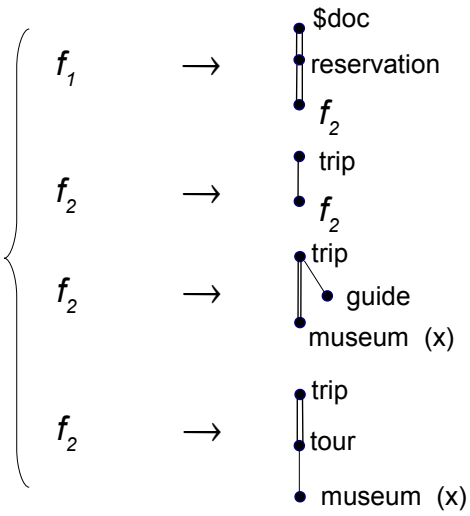


Figure 1.8: Example of Query Set Specification

A QSS has a set of rules, four in this case, similar to the rules of a grammar. The patterns in the rules have two types of nodes: tree fragment nodes (f_1 and f_2) and element nodes. f_1 is the start node, as all expansions of the QSS start by expanding f_1 (by the first rule) and continue until a pattern having only element nodes is obtained. Element nodes that have an output mark, drawn as "(x)", will become distinguished nodes in an expansion. (In Chapter 5 I use an equivalent syntax for QSS that is more appropriate for describing algorithms and proofs.)

For instance, v_1 from Figure 1.9 is obtained from the right hand side of the first rule by expanding f_2 using the second rule and then applying the third rule.

Suppose the user asks the query q from Figure 1.9 that looks for museums visited as part of tours on guided secondary trips. The adapter verifying support has to be able to find that, among the infinity of views generated by the QSS,

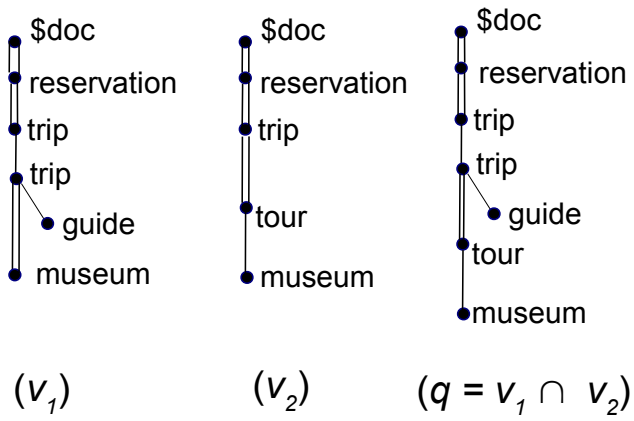


Figure 1.9: A query rewritten using two QSS generated views

there are also views v_1 and v_2 from Figure 1.9 such that q can be written as their intersection. This in turn proves that q is the supported by the QSS.

When studying expressibility and support for XPath queries and QSS encoded XPath views, I focused on obtaining efficient algorithms, as real applications need to compile user queries fast. In Chapter 5, I present the proofs of hardness for expressibility and support in the general case. Then I introduce restrictions that allow PTIME solutions for a large subset of the XPath language.

1.3 Outline

This dissertation studies expressibility and support both for listed and compactly encoded sets of services.

For listed (finite) sets of services, expressibility reduces to query equivalence, which has already been the object of several publications [MS04, DPX04]. Hence, for this case I focus on support and I analyze the problem of rewriting XML queries using views in the following two chapters. Chapter 2 is dedicated to rewriting nested XML queries using nested XML views. Chapter 3 studies the effect of adding intersection into the rewriting language for queries that perform only navigation (corresponding to the core of the XPath [CD99] language).

Chapter 4 treats expressibility and support for relational queries and (parameterized) views encoded as expansions of (parameterized) Datalog programs, under constraints on the data source. For a large class of queries, programs and constraints, I show that the two problems are decidable and present an algorithm that is in the same complexity class, EXPTIME, as the best algorithms for rewriting queries using a finite set of views.

Chapter 5 is dedicated to expressibility and support for XPath queries and views encoded as expansions of query set specifications (QSS [PDP03]). For classes of XPath queries defined in [CDO08], for which rewriting using views is tractable, I present syntactic conditions under which support by a QSS is tractable. I also show that relaxing these conditions leads to intractability.

Chapter 6 gives a brief overview of my work on building platforms that allow implementing the data services themselves. This includes extending the service infrastructure with XML update and scripting languages, integration with work flow languages, support for WSDL and optimizations across function calls.

Chapter 2

Rewriting XML Queries Using a Listed Set of XML Views

ABSTRACT OF THE CHAPTER

For listed sets of views, expressibility reduces to query equivalence, which has already been studied in depth [MS04, DPX04]. Therefore, in this chapter I will focus on support for listed views, which is traditionally called rewriting queries using views. I present and analyze an algorithm (which I first described in SIGMOD 2006 [ODPC06]) for equivalent rewriting of XQuery queries using XQuery views. The algorithm is complete for a large class of XQueries featuring nested FLWR blocks, XML construction and join equalities by value and identity. These features pose significant challenges which lead to fundamental extension of prior work on the problems of rewriting conjunctive and tree pattern queries. My solution exploits the Nested XML Tableaux (NEXT) notation which enables a logical foundation for specifying XQuery semantics. I present a tool which takes as input XQuery queries and views and outputs an XQuery rewriting, thus being usable on top of any of the existing XQuery processing engines. The experimental evaluation shows that the tool scales well for large numbers of views and complex queries.

Then I show how this algorithm can be extended to a sound procedure for the entire XQuery language. Giving up completeness cannot be avoided, as, for

arbitrary XQuery views and queries, the existence of a rewriting is undecidable.

2.1 Introduction

For a listed set of query-defined services, deciding if the user query is supported by the services is equivalent to deciding if the query has a rewriting using the queries that define the services. The queries that define services are public *views* [AHV95] over the data source. Thus, a solution for support in this setting is equivalent to a classical database problem, that of rewriting queries using views.

The ability to rewrite a query using views is required by multiple data management tasks. For example, a query processor can speed up the processing of a query when part of the computation needed by the query has already been performed by the materialized views or cached queries. Another application comes from the field of privacy-preserving data publishing, in which a data source answers only those client queries which can be rewritten using exclusively views the data owner agrees to publish [RMSR04]. Finally, in data integration views have been used to describe the source content (in Local-As-View architectures) and the source capabilities. We point the reader to the survey [Hal01] for a comprehensive list of applications of rewriting queries using views.

The XML data model emerges as equally important to the relational model for many of the data management problems that require answering queries using views. At the same time, solving the problem in the context of XML and XQuery presents a set of novel challenges. First, data and queries are nested. Second, XQuery has *list* semantics, which degenerates to *bag* semantics if the **unordered** keyword is used, and to *set* semantics in the presence of the duplicate-eliminating primitive **distinct-values** (as in Example 2.1.1 below). Any rewriting algorithm must be equipped to uniformly test the equivalence of the produced rewriting to the original query under any of these semantics. Contrast this with prior work on XPath (and relational) rewriting, which use only *set* semantics. Third, XML elements may be compared for equality on either their value or their identity; no such distinction comes up in conjunctive queries and XPath. Finally, XQuery

copies XML subtrees in the result construction phase, precluding the use of XML node ids for assembling the view data into the query result.

We provide an algorithm which, given an XQuery Q and a set of XQuery views $\mathcal{V} = \{V_1, \ldots, V_n\}$, discovers a rewriting query RW of Q using $\mathcal{V}$. We follow the classic definition of rewriting query [Hal01]: RW is evaluated on the views and for all possible database instances D its result $RW(V_1(D), \ldots, V_n(D))$ coincides with the result $Q(D)$ of Q. The algorithm is sound for full XQuery queries and views, and is complete for an expressive subset of XQuery called Opt-XQuery [DPX04], which includes nesting, optional use of duplicate elimination (using the **distinct-values** keyword), and allows in the **where** clause conjunctions of id-based and value-based equality conditions, as well as existential quantification (**some** clauses).

EXAMPLE 2.1.1. *Consider the sample data of Figure 2.1 and the following query Q that groups paper reviews by the papers' authors; it is a minor variation of query Q4 from W3C's XMP use case [CFF+]. The* **distinct-values** *function eliminates duplicates, comparing elements by value-based equality [BCF+].* [1] *The* **for** *loop binding* $a *(called the* $a *loop) has* set *semantics since the output list of* **distinct-values** *has no duplicates and the order of its elements is non-deterministic. The inner* **for** *loop has* bag *semantics i.e., duplicates are not removed but the order is non-deterministic since the loop is in the context of the unordered* $a *loop.*

```
let $doc := document("DBLP.xml")
for  $a in distinct-values($doc//paper[review]/author)
return ⟨evaluation⟩{ $a,                                          (Q)
       for  $p in  $doc//paper
            $r in  $p/review
       where  some  $a1 in  $p/author
              satisfies  $a1 eq $a
       return  $r
```

[1] The query can be expressed in a shorter form by replacing its **where** clause with "**where** $a **eq** $p/author" or by replacing the inner **for** with "$doc//paper[author **eq** $a]/review". It is well known [MFK01] how to reduce such syntactic sugar (use of "**eq** " or use of predicates in paths) to OptXQuery.

}⟨/evaluation⟩

Consider now the following view V, which outputs a list of feedback *elements, where each one contains a review and the list of authors of the corresponding paper.*

let *$doc* := **document***('DBLP.xml')*
for *$p* **in** *$doc//paper, $r* **in** *$p/review*
return ⟨feedback⟩{ *$r,* *(V)*
⟨authors⟩{ *$p/author* }⟨/authors⟩
}⟨/feedback⟩

Since the view V returns all author and review information which is pertinent to the query Q, there is a rewriting RW of Q using V:[2]

let *$doc* := **document***('V')*
for *$a2* **in distinct-values***($doc/feedback[review]/authors/author)*
return ⟨evaluation⟩{ *$a,* *(RW)*
for *$f* **in** *$doc/feedback*
$r **in** *$f/review*
where **some** *$a1* **in** *$f/authors/author*
satisfies *$a1* **eq** *$a*
return *$r*
}⟨/evaluation⟩

Our query processor discovers rewritings that use the views in order to obtain the variable bindings generated by the query.

To the best of our knowledge this is the first work on rewriting using views for XQuery, which also provides formal guarantees of completeness for a large subset of XQuery.

[2]We support views which output a collection of XML elements, rather than a well-formed document with a single root element (we generate an artificial root). For presentation simplicity, we do not show the navigation to the root element.

Prior work on rewriting using views for XML queries (reviewed in Section 2.9) focused on the XPath language [BÖB+04, XÖ05]. The resulting algorithms involve detecting query subexpressions subsumed by the view. The subsumption test is enabled by a pattern-based representation of XPath expressions called *tree patterns* [AYCLS02, MS04], which reduce the test to matching the view pattern against the query pattern. The standard specification of XQuery semantics does not support the extension of XPath rewriting techniques as it provides no pattern-based query representation. We therefore adopted *NEsted XML Tableaux (NEXT)* [DPX04], a pattern-based notation, which consolidates navigation in the minimum number of tree patters and hence maximizes rewriting opportunities. NEXT can represent a large subset of XQuery, called OptXQuery and introduced in [DPX04]. OptXQuery is a natural boundary within which the rewriting algorithm is guaranteed to find rewritings whenever they exist. The OptXQuery subset includes the XPath language, and it fully subsumes the c-XQuery subset of conjunctive, nested XQueries whose containment is studied in [DHT04a].

Our contributions are:

1. An algorithm for rewriting queries from the OptXQuery subset of XQuery using exclusively views from the same sublanguage. The algorithm equivalently rewrites the variable binding stage of the query, which is where the large costs of navigations, joins and selections of data intensive applications are incurred. If the document order of the query result is immaterial (due to the use of **distinct-values** and **unordered** keywords), then the algorithm is *complete*, i.e., it always finds an equivalent rewriting, if such exists (Theorem 2.4.1).

2. We analyze the complexity of checking the existence of a rewriting, showing NP-completeness in the *width* of the query's NEXT pattern representation. This measure (defined in Section 2.4.5), is typically much smaller than the query size, depending only on the number of variables shared across nested query blocks and the number of variables involved in equality conditions. For instance, the width of query Q in Example 2.1.1 is 3, while the query has 6 navigation steps. Our rewriting algorithm is worst-case exponential in the query width, which is optimal behavior: it runs in PTIME when the query is acyclic, (for which it turns

out that the width is 1) its performance degrading gracefully with increasing query width. The acyclic case includes XPath tree patterns.

3. We introduce a technique for extending the rewriting algorithm from OptXQuery to arbitrary XQuery, by identifying and rewriting the OptXQuery subexpressions. The technique is sound, in the sense that it creates only equivalent rewritings. It is not complete, but this is an unavoidable consequence of the undecidability of checking rewriting existence for (even slightly) more expressive queries and views than OptXQuery. Indeed, even relational rewriting algorithms perform a best-effort approach and are incomplete for full SQL. Our approach combines the benefits of complete rewriting feasible for NEXT/OptXQuery with the easy-to-engineer but incomplete techniques based on isomorphically matching common sub-expressions between query and view [DFK+04].

4. We allow the user to specify properties of XQuery functions and expressions such that more semantic information is captured and more rewritings are enabled.

5. We provide a tool which inputs XQuery queries, views and user defined function properties and outputs an XQuery rewriting, thus being usable on top of any of the existing XQuery processing engines. A demo of the tool is available at http://db.ucsd.edu/reform.

6. We report on our experimental evaluation, which measured rewriting times of around 1 second for as many as 128 views, for queries of up to 16 nesting levels, 48 variables per level.

Chapter outline. The remainder of this chapter is organized as follows. Section 2.2 describes the system architecture, OptXQuery and its corresponding NEXT notation. The translation of an OptXQuery/XQuery into a NEXT/NEXT+ form happens during normalization, described in Section 2.3. The normal form is the input of the rewriting algorithm presented in Section 2.4, together with its implementation and the complexity analysis. The extension of the rewriting algorithm to arbitrary XQueries, taking into account additional semantic properties, is given in Section 2.5. Section 2.6 explains additional changes needed to accommodate queries with ordered semantics. Optimizations that reduce the size of the

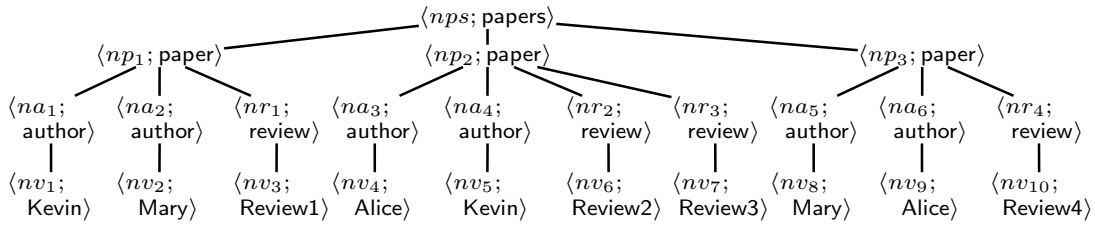


Figure 2.1: Input XML data

rewritten query plan are discussed in Section 2.7. We report on the experimental evaluation in Section 2.8. Related work is discussed in Section 2.9.

2.2 Architecture and Framework

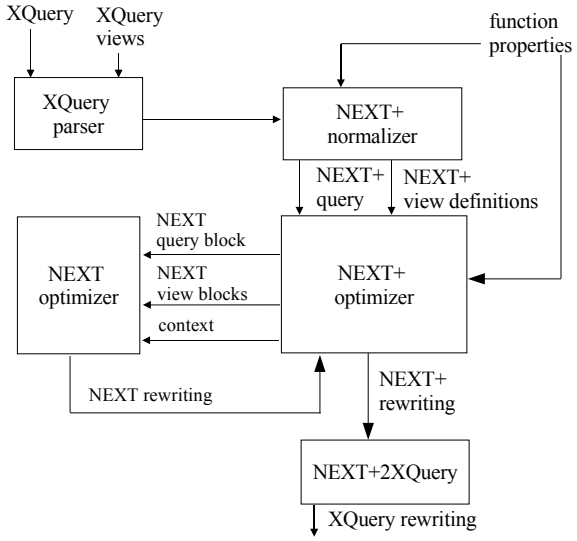


Figure 2.2: NEXT+ XQuery Processor Architecture

XML and Equivalence We model an XML document D as an ordered labeled tree of nodes $\mathtt{N}_{XML}$, edges $\mathtt{E}_{XML}$, a function $\lambda : \mathtt{N}_{XML} \rightarrow \mathit{Constants}$ that assigns a label to each node, and a function $: \mathtt{N}_{XML} \rightarrow \mathit{IDs}$ that assigns a unique

id to each node.

Equivalent rewriting. We start from the classic definition of the equivalent rewriting problem (taken from [Hal01]), adapting it for the XML case. Let Q be a query and $\mathcal{V} = \{V_1, \ldots, V_n\}$ be a set of view definitions. The query RW is an equivalent rewriting of Q using $\mathcal{V}$ if RW refers only to the views in $\mathcal{V}$ and for every input document D, Q and RW return isomorphic XML results: $Q(D)$ (the result of Q on D) is isomorphic to $RW(V_1(D), \ldots, V_n(D))$. We consider two flavors of isomorphism: unordered and ordered, leading to *ordered*, respectively *unordered rewritings*. Unordered isomorphism disregards the sibling ordering within each node, while ordered isomorphism preserves it. Unordered rewritings are appropriate whenever an XQuery contains a **distinct-values** or an **unordered** keyword, which cause the output XML tree to be non-deterministically ordered [BCF$^+$]. The **unordered** keyword is typically used in data-centric applications of XQuery (in which the document order is immaterial, and only the data contents matters). The **distinct-values** keyword is unavoidable in XQueries performing duplicate elimination or grouping by value, as illustrated in Example 2.1.1. When neither keyword appears in the query and the ordering of its answer XML tree is relevant, we resort to ordered rewritings.

Architecture The first stage in processing NEXT queries is represented by the normalization module [DPX04], which inputs a query and views expressed in XQuery, applies a series of normalization rules (described in detail in [DPX04]), and produces *Nested XML Tableaux (NEXT)* of the query and the views (see Figure 2.2). The NEXT representation consolidates all variable binding operations regardless of whether they appear in the **for** or the **where** clause and regardless of whether they are in the context of set (i.e., **distinct-values**) or bag semantics in the **in** clause. This consolidation facilitates the match of the navigation of the query with the navigation of the view. Not all XQuery expressions can be normalized into a NEXT representation. According to [DPX04], this is possible if the given XQuery follows the OptXQuery syntax of Figure 2.3 and refrains from set/list/bag equality comparisons [BCF$^+$]. We sketch in Section 2.5 how arbitrary XQueries are handled. This involved extending the NEXT notation to

$$
\begin{array}{rcl}
XQ & ::= & \langle n \rangle \{XQ_1, \ldots, XQ_m\} \langle /n \rangle \\
& | & XQ_1, XQ_2 \\
& | & \textbf{for } (V \textbf{ in } XQ)+ \ (\textbf{where } CList)? \textbf{ return } XQ \\
& | & (\textbf{document } (\text{“}Constant'')|\, Var) \ ((/|//)\, Constant)* \\
& | & Constant \\
& | & \textbf{distinct-values}(XQ) \\
& | & \textbf{unordered } (XQ) \\
CList & ::= & Cond \ (\textbf{and } Cond)* \\
Cond & ::= & Var_1 \textbf{ eq } (Var_2|Constant) \\
& | & Var_1 \textbf{is } Var_2 \\
& | & \textbf{some } (Var \textbf{ in } XQ)+ \ \textbf{satisfies } CList
\end{array}
$$

Figure 2.3: OptXQuery

accommodate all XQuery features, and detecting and rewriting only the maximal OptXQuery subexpressions. Additional semantic information can be captured by registering function properties, presented in Section 2.5.2, which can be used either at normalization or at optimization time in order to enable more rewritings.

The syntax of NEXT (see Figure 2.4) uses only a subset of OptXQuery features: (i) **in** clauses are (one-step) path expressions, (ii) **where** clause conditions are conjunctions of equality conditions involving variables and constants and (iii) there is no explicit **distinct-values**. These restrictions are compensated by a duplicate-eliminating projection operator, which in its full-blown version [DPX04] is a group-by construct. However, the grouping functionality is not needed by the rewriting algorithm. We keep the name **groupby** for consistency with the NEXT terminology and we advise the reader to think of **groupby** as duplicate-eliminating projection, as described below.

The query Q and the view V from Example 2.1.1 are rewritten into the NEXT normal forms below. NEXT extends FLWR expressions with a **groupby**

XQ ::= $\langle n \rangle \{ XQ_1, \ldots, XQ_m \} \langle /n \rangle$ (P1)

| Var (P2)

| **for** Var_1 **in** $Path_1, \ldots, Var_n$ **in** $Path_n$ (P3)

(**where** $CList$)?

groupby $(Var'_1 | [Var'_1]) \ldots (Var'_k | [Var'_k])$

return XQ_1

$Path$ ::= (**document** ("$Constant''$)| Var)((/|//)$Constant$) (P4)

$CList$::= $Cond$ (**and** $Cond$)$*$ (P5)

$Cond$::= Var **eq** (Var|$Constant$) (P6)

|Var **is** Var (P7)

Figure 2.4: NEXT Query Syntax

clause that consists of a list of group-by variables. A variable from the **groupby** list is called *groupby-id variable* if it appears within brackets (e.g., variable $\$p1$ of (NEXT_Q)), and is called *groupby-value variable* otherwise (e.g, variable $\$a$ of (NEXT_Q)). A **groupby** construct inputs the tuples of variable bindings produced by the preceding **for** and **where** clauses and projects the variables of the **groupby** list, eliminating duplicates. To detect duplicates, tuples are compared component-wise, looking for equal values on the bindings of the groupby-value variables and for equal id's for the groupby-id variables. The **return** clause is executed once for each tuple in the output of the **groupby** clause.

B_1^Q {
for $\$p$ **in document**("$DBLP.xml''$")//$paper$,
  $\$r$ **in** $\$p/review$, ($NEXT_Q$)
  $\$a$ **in** $\$p/author$
groupby $\$a$
return ⟨evaluation⟩{ $\$a$,
  B_2^Q {
    for $\$p1$ **in document**("$DBLP.xml''$")//$paper$,
      $\$a1$ **in** $\$p1/author$,
      $\$r1$ **in** $\$p1/review$
    where $\$a1$ **eq** $\$a$
    groupby [$\$p1$], [$\$r1$]
    return $\$r1$
  }⟨/evaluation⟩
}

B_1^V {
for $\$p$ **in document**("$DBLP.xml''$")//$paper$,
  $\$r$ **in** $\$p/review$ ($NEXT_V$)
groupby [$\$r$], [$\$p$]
return ⟨feedback⟩{ $\$r$,
  ⟨authors⟩{
  B_2^V {
    for $\$a$ **in** $\$p/author$
    groupby [$\$a$]
    return $\$a$
  }⟨/authors⟩
  }⟨/feedback⟩
}

The tree of **groupby** *blocks.* We call **groupby** *block* a NEXT expression described by Production (P3) of Figure 2.4. For example, NEXT_Q contains blocks B_1^Q and B_2^Q as illustrated above. If a block B' is immediately (transitively) nested within block B, we say that B' is a child (descendant) of B and B a parent (ancestor) of B'. The **groupby** blocks of a query are therefore organized in a tree structure, called the **groupby** tree. The *bound variables* of a block B are those variables $\$v$ appearing in a "$\$v$ **in** *path*" expression. All other variables are *free* in B, but must be bound in some ancestor block of B. For instance, $\$a$ is bound in B_1^Q and free in B_2^Q.

Pattern-based graphical representation of NEXT. NEXT queries have a

graphical, pattern-based notation that extends the well-known tree patterns used in XPath-related works. The representation depicts the tree of groupby blocks, labeling each block with a pattern, a list of groupby-value variables, a list of groupby-id variables, and a return function. A NEXT pattern $P = (F, EQ_{val}, EQ_{id})$ consists of a forest F of tree patterns, which capture navigation. As is common in tree pattern notations, each node is labeled with (i) a variable and (ii) the label that this node matches to. Edges are labeled with / or // depending on whether the corresponding nodes are in a child or descendant relationship. EQ_{val} is the set of value-based equalities, denoted by dotted lines, and EQ_{id} is the set of id-based equalities (denoted by double dotted lines, though not needed in our example). The bound variables of a NEXT pattern are precisely the variables which are the target of some /- or //-edge. The return functions consist of element creation and concatenation and take as parameters variables and nested blocks. In the latter case, the meaning is that, upon instantiation, the parameter is replaced by the result of the corresponding block.

EXAMPLE 2.2.1. *Figure 2.5 shows the NEXT representation for Q and V, and Figure 2.6 shows the same for RW. In each case, we see the tree of* **groupby** *blocks on the left, followed by the corresponding NEXT patterns, and to their right, the return functions. The second argument of the groupby blocks shows the list of groupby-value variables ($\$a$ for B_1^Q, empty for B_2^Q, B_1^V and B_2^V). The third argument gives the list of groupby-id variables (empty for B_1^Q, $\$p_1, \$r_1$ for B_2^Q).*

Understanding of the rewriting algorithm is facilitated if one thinks of the result of NEXT query Q on an XML document D as being computed in two stages. First, the *binding stage* computes all variable bindings by matching the NEXT patterns against the XML input. Bindings are organized in a nested relation according to the nesting of **groupby** blocks. We call this nested table the *binding table*. Next, the *tagging stage* operates on the binding table, constructing an XML fragment for each tuple in the nested relation by calling the appropriate return function. Denoting with Q^{bind} the function (from XML documents to binding tables) implemented by the binding stage, and with Q^{tag} the function (from binding tables to XML documents) implemented by the tagging stage, we have

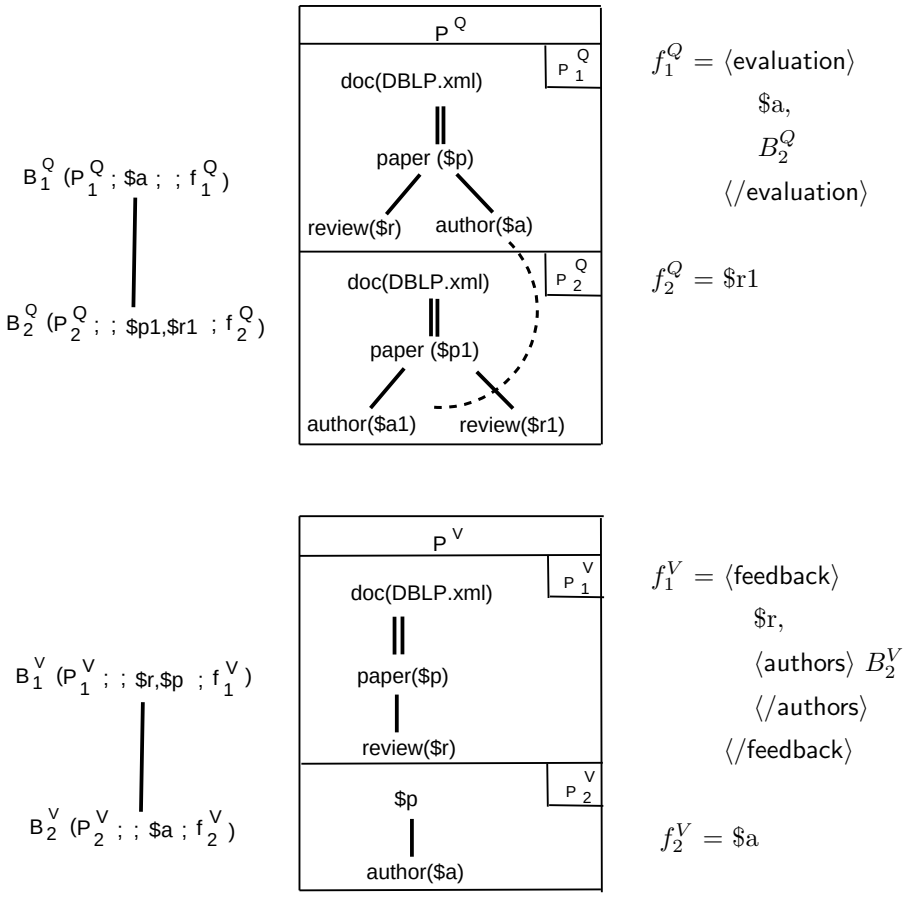


Figure 2.5: NEXT for Q and V from Example 2.1.1

$Q^{tag}(Q^{bind}(D)) = Q(D)$ for every XML document D and NEXT query Q.

EXAMPLE 2.2.2. *For the sample input provided in Figure 2.1, $NEXT_Q$'s binding stage yields the nested relation of variable bindings shown in Table 2.1. Notice that the binding table's attribute names coincide with the variable names and the names of the nested* **groupby** *blocks. For each tuple, the attributes named after variables hold the corresponding bindings of groupby variables and the attributes named after the nested* **groupby** *blocks recursively hold the corresponding set of bindings, which is in turn a nested relation.*

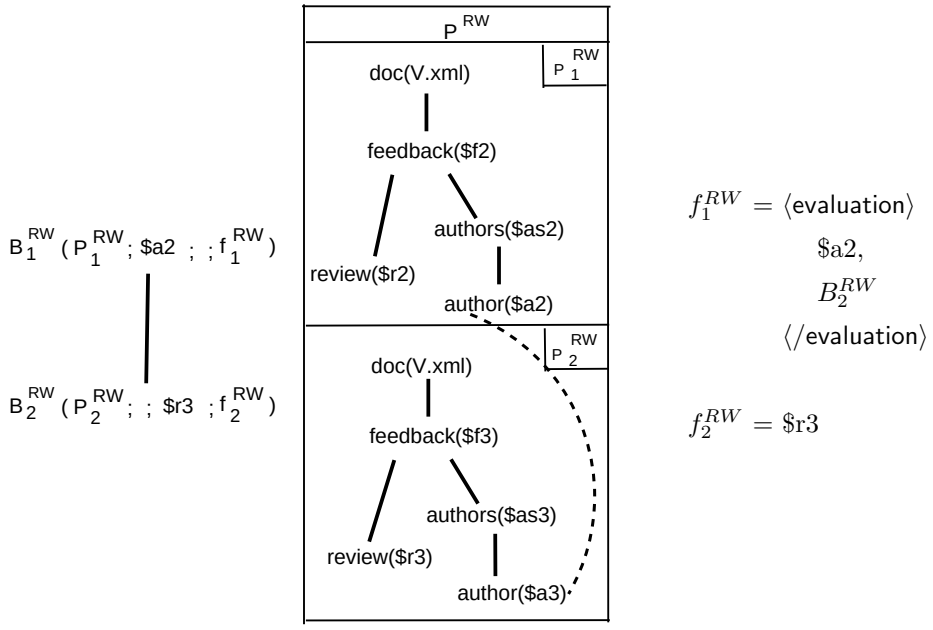


Figure 2.6: NEXT for RW from Example 2.1.1

Table 2.1: Result of NEXT$_Q$'s binding stage

B_1^Q		
$\$a$	B_2^Q	
<author> Kevin </author>	$[\$p1]$	$[\$r1]$
	np_1	nr_1
	np_2	nr_2
	np_2	nr_3
<author> Mary </author>	$[\$p1]$	$[\$r1]$
	np_1	nr_1
	np_3	nr_4
<author> Alice </author>	$[\$p1]$	$[\$r1]$
	np_2	nr_2
	np_2	nr_3
	np_3	nr_4

2.3 Normalization

The transition from the XQuery syntax, complex and hard to reason about, to the NEXT form, simpler and more amenable to optimizations, is performed by

a process of normalization based on rewriting rules[3]. Our normalizer comprises a sequence of stages (See Figure 2.7) and a list of rules is associated to each stage. Within the same stage, each rule from the list is repeatedly invoked until it is no longer applicable, then the next rule is invoked. This execution process stops when there is no rule applicable after trying the rules from beginning to end; otherwise, the list of rules is run again. This process is necessary since a rule may create rewriting opportunities for other rules previously tried and found not applicable.

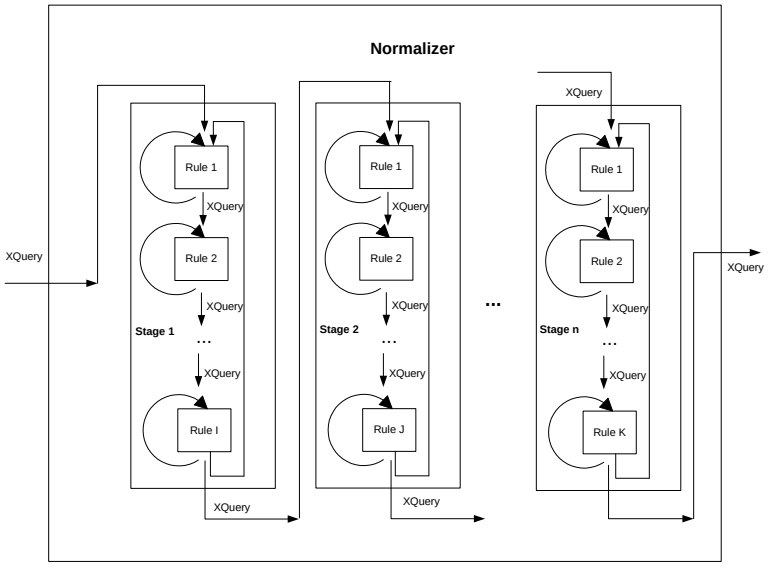


Figure 2.7: Stage based normalization

Table 2.5 presents a set of rewrite rules which provably normalize any XQuery to a NEXT query (as shown by Theorem 2.3.1). Some of these rules are known simplification rules of XQuery; they are used extensively both in reducing XQuery to its formal core [DFF$^+$] and in query optimization [MFK01].

The normalization process is stratified in two stages. First, all standard XQuery rewriting rules are applied in any order. Next, the **groupby** specific rules are used. A special case is represented by rule (RG1) that may be applied in either stage.

[3]Parts of the normalization of NEXT patterns appeared in [DPX04] and in Yu Xu's PhD dissertation [Xu05].

EXAMPLE 2.3.1. *Recall query Q from Example 2.1.1. In the first phase of its normalization, rules (R18), (R12), (R13) and (R20) apply, yielding the query below:*

```
for  $a in distinct-values($doc//paper[review]/author)
    for  $p in  document("DBLP.xml")//paper
    where  (for $r in  $p/review return  $r)
    return (for $a' in  $p/author return  $a')
return  ⟨evaluation⟩{ $a,
        for  $p in  document("DBLP.xml")//paper
        return
            for  $r in  $p/review
        where  some  $a1 in  $p/author
                satisfies  $a1 eq $a
        return  $r
        }⟨/evaluation⟩
```

The second normalization phase applies **groupby** *rewriting rules to this intermediate result. A rewrite step with Rule (G1) applied to the outermost* **for** *replaces the* **distinct-values** *function with a* **groupby** *clause which groups by the value of variable* $a*. Similarly, Rule (G4) turns the inner* **for** *expressions, which do not involve* **distinct-values***, into a* **for** *expression that involves grouping by identity. By applying rule (G5) the* **some** *constructs are turned into* **for** *clauses whose bound variables do not participate in the groupby variable lists. Rule (G7) inlines subqueries nested in* **in** *clauses and rule (G8) inlines unnecessary* **for** *clauses. (G11) collapses* **groupby** *'s. The final result is exactly the query* $NEXT_Q$.

In order to have a proper normal form, we have to guarantee that

- normalization always terminates
- we always obtain the same effect as in example 2.3.1, i.e. for any OptXQuery, the normalization module finds an equivalent NEXT query.

From the syntactic point of view, a query Q is not an OptXQuery if it contains one of the following expressions which in general cannot be rewritten into NEXT form (See Figure 2.4):
"**some** $\$V$ **in** (E_1, E_2) **satisfies** C", "**distinct-values**(E_1, E_2)", "$\langle e\rangle\{E\}\langle/e\rangle//c$", "$\langle e_1\rangle\{E_1\}\langle/e_1\rangle$ **eq** $(\$V|\langle e_2\rangle\{E_2\}\langle/e_2\rangle|Constant)$".
Although the last two types of expressions are not allowed in the grammar shown in Figure 2.3, both of them could appear after application of some rewriting rules in Figure 2.5 (R2 and R4).

However, syntactic constraints are not enough to enforce that only NEXT queries are obtained after normalization. We need additional constraints and in order to express them, we will introduce additional notations. The *definition* $def(\$V)$ of a variable \$V in "$\$V$ **in** E" is denoted $def(\$V) = f(E)$ where $f(E) = E$ if E is $\langle c\rangle\{XQ\}\langle/c\rangle$, or a path expression; $f(E) = f(E_1)$ if E is **distinct-values**(E_1); $f(E_1, E_2) = f(E_1), f(E_2)$; $f(E) = f(E_r)$ if E is a FLWR expression and E_r is the return clause of E. Note the recursive definition in the case of FLWR expressions.

A variable $\$V$ *directly depends* on $\$V'$ if $\$V'$ appears in $def(\$V)$. We say that $\$V$ *depends* on $\$V'$ if it directly or indirectly (via other variables) depends on $\$V'$. A variable \$V is called a *basic variable* if the definition of $\$V$ and the definition of any variable that $\$V$ depends on does not contain element constructor or concatenation. An element constructor $\langle e\rangle\{E\}\langle/e\rangle$ is called a *simple element constructor* if the element $\langle e\rangle$ is created by the query via repeated application of element constructors to constants, simple variables and simple element constructors. Note the recursive definition and the fact that complex expressions such as path expressions or **for** loops are disallowed. A variable \$V is called a *simply constructed variable* if $def(\$V)$ is a simple element constructor.

A variable \$V whose definition is "$\$X/c_1/.../c_n$" is called a *simple path variable* if all of the following conditions are met: \$X is a simply constructed variable; every navigation step in the path expression is "/"; the result of normalizing $def(\$X)/c_1/.../c_n$ (Rule RG1) is a basic variable; and every intermediate result of evaluating $def(\$X)/c_1$, $def(\$X)/c_1/c_2$, ..., $def(\$X)/c_1/.../c_{n-1}$ is a basic variable or a simple element constructor. Notice that a simple path variable binds to single

elements from the input just like a basic variable. Therefore, we say that basic variables and simple path variables together are *input element variables*.

We exemplify the different types of variables on query Q' below that is grouping papers by their titles and the conference in which they were published.

```
for $r in distinct-values(
      for $p1 in $doc//paper,
          $t1 in $p1/title, $c1 in $p1/conference
      return ⟨ref⟩⟨t⟩{$t1}⟨/t⟩⟨c⟩{$c1}⟨/c⟩⟨/ref⟩),
   $t in $r/t/title, $c in $r/c/conference                    (Q')
return ⟨result⟩ {$t}{$c}
                { for $p' in $doc//paper
                  where some  $t' in $p'/title,
                              $c' in $p'/conference
                  satisfies $t' eq $t and $c' eq $c
                  return $p'/title}
        ⟨/result⟩
```

In Q', $\$p_1$, $\$t_1$, $\$c_1$, $\$p'$, $\$t'$, $\$c'$ are basic variables; $\$r$ is a simply constructed variable; $\$t$ and $\$c$ are simple path variables, but not basic variables since they depend on $\$r$, a simply constructed variable, whose definition contains element constructors.

Now we have all the infrastructure to formulate the constraints $\mathscr{C}$, listed in Table 2.2, under which normalization of OptXQuery is complete. The first four constraints guarantee that none of the four types of non-NEXT expressions mentioned above would appear while applying the normalization rules. The purpose of the last constraint is to rule out constructs that would simulate an **if** expression (because **if** is not part of the NEXT grammar).

Lemma 2.3.1. *The rewriting of an OptXQuery with the normalization rules in Table 2.5 terminates.*

Proof:

Table 2.2: The $\mathscr{C}$ constraints for OptXQuery

(C_1)	for each \$V defined by "$\$V$ **in** E" in a **some** clause, \$V and every variable \$X in E and every variable that \$X depends on are basic variables.
(C_2)	For each **distinct-values**(E), if we define \$V as "$\$V$ **in distinct-values**(E)", \$V must be an input element variable or simply constructed variable; for each path expression "$\$V(/\|//)c_1 \ldots (/\|//)c_n$" where \$V depends on a variable \$X defined in "$\$X$ **in distinct-values**(E), if we define $\$V_i$ as "$\$V_i$ **in** $\$V(/\|//)c_1 \ldots (/\|//)c_i$", $\$V_i$ must be a simply constructed variable or an input element variable $(i = 1, \ldots, n)$.
(C_3)	If a path contains a "//" step and starts from a variable $\$V$, then $\$V$ is an input element variable.
(C_4)	All variables appearing in equality conditions are input element variables.
(C_5)	No "$\$V$ **in** $\$V'$" is allowed; at least one variable $\$V_i$ in "**for** $\$V_1$ **in** $E_1, \ldots, \$V_n$ **in** E_n **where** C **return** E" must be a basic variable.

Consider the first rewriting stage. We associate a vector τ, $\langle sp, mp, var, elm, con, vp, some, dis, dm\rangle$, with each query Q, with sp being the most significant part of τ. Intuitively each component of τ indicates the degree the query violates the NEXT form in some aspect. We choose τ such that each rule decreases its value, considering a lexicographic order of its components, and, as it will become apparent from the definition, all components take positive values only. Thus we will have a proof that the first stage of normalization always terminates.

Note that we need to pay special attention to certain rules when defining

$$
\begin{aligned}
XQ &::= \langle n\rangle\{XQ_1,\ldots,XQ_m\}\langle/n\rangle \mid FLWR \mid Constant \\
&\mid\ XQ_1, XQ_2 \mid \textbf{distinct-values}\ (FLWR) \\
FLWR &::= \textbf{for}\ (V\ \textbf{in}\ SP)+\ (\textbf{where}\ CList)?\ \textbf{return}\ XQ \\
SP &::= Path \mid \textbf{distinct-values}\ (FLWR) \\
Path &::= (\textbf{document}\ (\text{“}constant''") \mid Var)(/|//)Constant \\
CList &::= Cond\ (\textbf{and}\ Cond)* \\
Cond &::= Var_1\ \textbf{eq}\ (Var_2 \mid Constant) \\
&\mid\ \textbf{some}\ V\ \textbf{in}\ Path\ \textbf{satisfies}\ CList
\end{aligned}
$$

Figure 2.8: XQuery Normal Form

τ. Such a delicate case is, for instance, rule (R2), which substitutes \$V with $\langle e\rangle\{E_1\}\langle/e\rangle$ inside another expression E_2. E_1 may not be in the NEXT normal form, yet E_2 might contain multiple occurrences of \$V and be normalized. After substitution, τ may increase if it is not properly designed.

To simplify the presentation, we require that input queries have no variable defined more than once, which can easily be achieved by variable renaming. The components of the vector τ of a query Q are defined as:

- *sp*, the number of "$\$V(/|//)c$" expressions in Q, except for the ones appearing inside **in**-clauses of the form "$\$X$ **in** $\$V(/|//)c$".
- *mp*, the number of expressions "$\$V(/|//)c_1 \ldots /|//c_n$", $n \geq 2$, in Q.
- *var*, the number of variables defined inside **in**-clauses in Q
- *elm*, the number of *direct element variables*. $\$V$ defined by "$\$V$ **in** E" is a direct element variable in two cases. The first one is if E is an element constructor or another direct element variable. The second case is when E is a concatenation expression and one of its components is an element constructor or another direct element variable.

Table 2.3: The impact of rewriting rules (applicable for OptXQuery) on τ

	sp	mp	var	elm	con	vp	some	dis	dm
R1	- ↓	- ↓	↓	-↑	- ↓	- ↓	-	-	-
R2	- ↓	- ↓	- ↓	↓	- ↓ ↑	- ↓↑	- ↓ ↑	- ↓ ↑	- ↓↑
R3	-	-	- ↓	- ↓	-	↓	-	-	-
R4	- ↓	- ↓	- ↓	↓	- ↓ ↑	- ↓↑	- ↓ ↑	- ↓ ↑	- ↓↑
R5	-	-	- ↓	- ↓	-	↓	-	-	-
R6	- ↓	-	-	-	↓	- ↑	- ↑	- ↑	-↑
R7	-	-	-	-	-	-	↓	-	-
R8	-↓	-	↓	-↑	-↓	-↓	-	-	-
R9	-↓	-↓	- ↓	↓	- ↓ ↑	-↓ ↑	-↓ ↑	-↓ ↑	-↓ ↑
R10	-	-	- ↓	- ↓	-	↓	-	-	-
R11	-	-	-	-	-	-	-	↓	-
R12	↓	-	-↑	-	-	-	-	-	-
R13	-	↓	-↑	-	-	-	-	-	-
R14	-	-	-	-	-	-	-	↓	-
RG 1	-↓	-↓	-↓	- ↓	-↓	-↓	-↓	- ↓	↓

- *con*, the number of concatenations inside **in**-clauses in Q
- *vp*, the number of "$\$V$ **in** E" where E has not the form "$\$X(/|//)c$".
- *some*, the number of **some** clauses that define more than one variable anywhere in Q.
- *dis*, the number of occurrences of the **distinct-values** function anywhere in Q.
- *dm* is the number of "$\langle e\rangle\{E\}\langle/e\rangle/c$" occurrences in Q.

For instance, for our examples, $\tau(Q') = \langle 1,2,3,0,3,3,1,1,0\rangle$, $\tau(Q) = \langle 1,1,0,0,0,1,0,1,0\rangle$.

Table 2.3 shows how each rule in Table 2.5 may change τ where ↑ means increase, ↓ decrease, - no change. As it can be seen, the application of any first stage rule makes τ decrease.

Table 2.4: The impact of rewriting rules (applicable for OptXQuery) on υ

	dis	wogb	somec	gbc	invar	gbvar
G1	$\downarrow$	$\downarrow$	-	$\uparrow$	-	$\uparrow$
G2	$\downarrow$	$\downarrow$	-	$\uparrow$	-	$\uparrow$
G3	$\downarrow$	- $\downarrow$	-	$\uparrow$	-	$\uparrow$
G4	-	$\downarrow$	-	$\uparrow$	-	$\uparrow$
G5	-	-	$\downarrow$	-	- $\downarrow$	-
G6	-	$\downarrow$ -	$\downarrow$	$\uparrow$ -	- $\downarrow$	$\uparrow$ -
G7	-	-	-	$\downarrow$	- $\uparrow$	$\downarrow$
G8	-	-	-	$\downarrow$	$\downarrow$	$\downarrow$
G9	-	-	-	$\downarrow$	$\downarrow$	$\downarrow$
G10	-	-	-	- $\downarrow$	$\downarrow$	- $\downarrow$
G11	-	-	-	$\downarrow$	-	-
G12	-	-	-	-	-	$\downarrow$

Similarly, we can prove that the second normalization stage terminates, by associating to each query the vector $\upsilon = \langle dis, wogb, somec, gbc, invar, gbvar \rangle$. The meaning of each component is:

- *dis*, same as in τ
- *wogb*, number of FLWRs without any **groupby** clause
- *somec*, number of **some** clauses
- *gbc*, number of **groupby** clauses
- *invar*, number of **in**-clauses consisting in a variable reference (e.g. $\$V$ **in** $\$V'$)
- *gbvar*, number of **groupby** variables

Table 2.4 shows how the application of every second stage rule makes υ decrease, when considering lexicographic order comparison.

■

Lemma 2.3.2. *If the constraints $\mathscr{C}$ are satisfied, then the result of the normalization of an OptXQuery with the rules in Table 2.5 is a NEXT query.*

Proof:

Lemma 2.3.2 guarantees that normalization terminates. We also show that the final result is a NEXT query and in order to do this we will first show that after the first normalization stage an OptXQuery Q is rewritten into Q' which is in the XNF form defined in Figure 2.8.

Syntactically, there are three differences between OptXQuery and XNF. First, the **some** expressions in an OptXQuery may define more than one variable. Second, path expressions in OptXQuery may have more than one step or appear outside of variable definitions. Third, variables in an XNF query can only be defined by single step path expressions or, in addition, by calls to **distinct-values**, if bound by **for** loops.

We can prove, by contradiction, that Q' must be in the XNF form. If the first type of violation of XNF appeared, rule R7 would be applicable. If the second type of violation of XNF form existed, rules R12 or R13 would be applicable. If the third type of violation of XNF form existed, consider a variable defined in "**for** $\$V$ **in** E" loop. If E is a FLWR expression, an element constructor, a variable, a concatenation expression, a path expression of more than one step or $\langle e\rangle\{E_1\}\langle/e\rangle/c$, Rules R1, R2, R5, R3, R6, R13, RG1, are applicable, which contradicts the assumption that the rewriting has terminated. Notice that $\langle e\rangle\{E_1\}\langle/e\rangle/c$ is not allowed to define variables in OptXQueries but may appear in the definition of a variable after application of Rule R2, and it is the only type of expression that may be introduced by rewriting as the definition of a variable because of Rule R2 and R4. Notice that $\langle e\rangle\{E_1\}\langle/e\rangle//c$ cannot appear during rewriting because of the constraint C_3. Similarly we can show that if a variable is bound to an expression E in an expression "**some** $\$V$ **in** E", then E can only be a single step path expression and it cannot be a call to the **distinct-values** function because of Rule R11. Since variables in equality conditions are required to be input element variables, equality conditions would not be affected (by Rules R2 and R4) and are still of the form "$\$V$ **eq** $\$V'$" or "$\$V$ **eq** c" after rewriting.

The second normalization stage rewrites a query in XNF form into a NEXT query. Syntactically, there are three differences between the XNF and the NEXT

form. First, a query Q in XNF may have calls to **distinct-values**, but they are eliminated by Rules G1 and G3. Second, Q does not have **groupby** clauses. Rules G1-G4 add **groupby** clauses to all FLWR expressions from Q, turning them into *FLWGR expressions*. Third, Q may have **some** clauses which Rules G5 and G6 eliminate. When the rewriting terminates, each E from an **in** clause "$\$V$ **in** E" can only be a single step path expression, which again can be proven by contradiction. If E were a FLWGR expression, an element constructor or a variable, Rules G7, G8, G9, G10 would be applicable, which contradicts the assumption that the rewriting terminated. And because (C_5) holds, no conditional expression needs to be introduced (as it can happen with non-OptXQuery expressions, for which rule G13, defined in Appendix 2.A, would be applicable).

■

Now we are able to prove the main normalization result:

Theorem 2.3.1. *The rewriting of any OptXQuery Q with the rules in Table 2.5 terminates regardless of the order in which rules are applied, i.e. a rewritten query T is obtained for which no rewrite rule applies. If the constraints $\mathscr{C}$ are verified, then T is guaranteed to be a NEXT query.*

Proof: Follows from lemmas 2.3.1 and 2.3.2. ■

Table 2.5: OptXQuery Normalization Rules

Standard XQuery Rewriting Rules

(We use $\mathbb{B}$ (with or without subscript) to denote a list of **for** clauses.)

(R1)

for $\mathbb{B}_1, \$V_1$ **in** (**for** $\mathbb{B}_2$(**where** C_1)? **return** E_1), $\mathbb{B}_3$ (**where** C_2)?
return E_2
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}_2, \$V_1$ **in** $E_1, \mathbb{B}_3$**where** C_1 **and** C_2 **return** E_2

(R2)

for $\mathbb{B}_1, \$V$ **in** $\langle e \rangle \{E_1\} \langle /e \rangle, \mathbb{B}_2$
($\mathbf{where}\ C$)? **return** E_2
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}'_2$
($\mathbf{where}\ C'_1$)? **return** E'_2
(* $\theta_{\$V \mapsto E_1}(E_2)$ substitutes E_1 for $\$V$ in E_2 *)
$C' = \theta_{\$V \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}(C)$
$E'_2 = \theta_{\$V \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}(E_2)$

(R3)

for $\$V_1$ **in** $\$V_2$ **return** E
$\mapsto$
$\theta_{\$V_1 \mapsto \$V_2}(E)$ (*if $\$V_2$ is not defined by **let** *)

(R4)

for $\mathbb{B}$ **return for** $\$V$ **in** $\langle e \rangle \{E_1\} \langle /e \rangle$ **where** C **return** E_2
$\mapsto \theta_{\$V \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}$(**for** $\mathbb{B}$ **where** C **return** E_2)

(R5)

for $\mathbb{B}$ **return for** $\$V_1$ **in** $\$V_2$ **where** C **return** E_1
$\mapsto \theta_{\$V_1 \mapsto \$V_2}$(**for** $\mathbb{B}$ **where** C **return** E_1) (*if $\$V_2$ is not defined by **let** *)

(R6)

for $\$V$ **in** (E_1, E_2) ($\mathbf{where}\ C$)? **return** E
$\mapsto$
$\theta_{\$V \mapsto \$V_1}$(**for** $\$V_1$ **in** E_1($\mathbf{where}\ C$)? **return** E),
$\theta_{\$V \mapsto \$V_2}$(**for** $\$V_2$ **in** E_2($\mathbf{where}\ C$)? **return** E)

(R7)

some $\$V_1$ **in** $E_1, \ldots, \$V_n$ **in** E_n **satisfies** C
$\mapsto$
some $\$V_1$ **in** E_1 **satisfies**
  some $\$V_2$ **in** E_2 **satisfies**
    ...
      some $\$V_n$ **in** E_n **satisfies** C

(R8)
some $\$V$ **in** (**for** $\$V_1$ **in** E_1 **return** E_2) **satisfies** C
$\mapsto$
some $\$V_1$ **in** E_1 **satisfies some** $\$V$ **in** E_2 **satisfies** C

(R9)
some $\$V$ **in** $\langle e\rangle\{E_1\}\langle/e\rangle$ **satisfies** C
$\mapsto$
$\theta_{\$V\mapsto\langle e\rangle\{E_1\}\langle/e\rangle}(C)$

(R10)
some $\$V_1$ **in** $\$V_2$ **satisfies** C
$\mapsto$
$\theta_{\$V_1\mapsto\$V_2}(C)$ (* if $\$V_2$ is not defined by **let** *)

(R11)
some $\$V$ **in distinct-values** (E) **satisfies** C
$\mapsto$
some $\$V$ **in** E **satisfies** C

(R12)
$\$V(/|//)C \mapsto$ **for** $\$V_1$ **in** $\$V(/|//)C$ **return** $\$V_1$
(* if $\$V/C$ does not appear in "$\$X$ **in** $\$V/C$"*)

(R13)
$\$V(/|//)C_1 \ldots (/|//)C_n$
$\mapsto$
for $\$V_1$ **in** $\$V(/|//)C_1, \ldots, \$V_n$ **in** $\$V_{n-1}(/|//)C_n$ **return** $\$V_n$
(* for $n \geq 2$ *)

(R14)
distinct-values $(\$V|\langle e\rangle\{E_1\}\langle/e\rangle|$**distinct-values** $(E))$
$\mapsto$
$\$V|\langle e\rangle\{E_1\}\langle/e\rangle|$**distinct-values** (E)
(*if $\$V$ is not defined by **let** *)

$\langle e\rangle\{E_1,\ldots,E_n\}\langle/e\rangle/c \mapsto \sigma_c(E_1),\ldots,\sigma_c(E_n)$

$\sigma_c(\langle c\rangle\{E\}\langle/c\rangle) \mapsto \langle c\rangle\{E\}\langle/c\rangle$

$\sigma_c(\langle a\rangle\{E\}\langle/a\rangle)\ (*a \neq c*) \mapsto ()$

$\sigma_c(\$V) \mapsto \$V \quad (*if(node-name(\$V) = c)*) \quad () \quad (*else*)$

$\sigma_c(E(/|//)a) \mapsto ()(*a \neq c*)$

(RG1) $\sigma_c(\textbf{for}\ \mathbb{B}\ (\textbf{where}\ C)?\ \textbf{return}\ E_2)$

$\mapsto$

$\textbf{for}\ \mathbb{B}\ (\textbf{where}\ C)?\ \textbf{return}\ \sigma_c(E_2)$

$\sigma_c(E(/|//)c) \mapsto E(/|//)c$

$\sigma_c(E_1, E_2) \mapsto \sigma_c(E_1), \sigma_c(E_2)$

$\sigma_c(\textbf{distinct-values}(E)) \mapsto \textbf{distinct-values}(\sigma_c(E))$

Groupby Rewriting Rules

$\textbf{for}\ \$V\ \textbf{in}\ \textbf{distinct-values}(E_1)\ (\textbf{where}\ C)?\ \textbf{return}\ E_2$

(G1) $\mapsto$

$\textbf{for}\ \$V\ \textbf{in}\ E_1\ (\textbf{where}\ C)?\ \textbf{groupby}\ \$V\ \textbf{return}\ E_2$

$\textbf{for}\ \mathbb{B}_1, \$V\ \textbf{in}\ \textbf{distinct-values}(E_1), \mathbb{B}_2(\textbf{where}\ C)?\ \textbf{return}\ E_2$

$\mapsto$

(G2) $\textbf{for}\ \mathbb{B}_1\textbf{return}$

$\quad \textbf{for}\ \$V\ \textbf{in}\ E_1\ \textbf{groupby}\ \$V\ \textbf{return}$

$\quad\quad \textbf{for}\ \mathbb{B}_2(\textbf{where}\ C)?\ \textbf{return}\ E_2$

(G3) $\textbf{distinct-values}(E_1) \mapsto \textbf{for}\ \$V\ \textbf{in}\ E_1\ \textbf{groupby}\ \$V\ \textbf{return}\ \$V$

(*if **distinct-values**(E_1) does not appear in "$\$X$ **in distinct-values** (E_1)"*)

$\textbf{for}\ \mathbb{B}\ (\textbf{where}\ C)?\ \textbf{return}\ E$

$\mapsto$

(G4) $\textbf{for}\ \mathbb{B}\ (\textbf{where}\ C)?\ \textbf{groupby}\ [\vec{B}]\textbf{return}\ E$

(* If all variables in $\vec{\mathbb{B}}$ are bound to sequences that have no duplicates by id. $\vec{\mathbb{B}}$ is the list of variables defined in $\mathbb{B}$ that are visible to E. In practice, these are checked by verifying whether the expressions that variables in $\vec{\mathbb{B}}$ bind to are one of the following types: path expressions that include a"/" or"//" operator or an axis step; union, intersect, and except expressions. *)

(G5)

for $\mathbb{B}$ **where** $((C_1$ **and** $)?$ (**some** $\$V$ **in** E_1 **satisfies** $C_2)$ $(\textbf{and}\ C_3)?)$
groupby G **return** E_2

$\mapsto$

for $\mathbb{B}, \$V$ **in** E_1 **where** C_1 **and** C_2 **and** C_3 **groupby** G
return E_3

(*if G contains only grouping-by-value expressions*)

(G6)

for $\mathbb{B}$ **where** $((C_1$ **and** $)?$ (**some** $\$V$ **in** E_1 **satisfies** $C_2)$ $(\textbf{and}\ C_3)?)$
return E_2

$\mapsto$

for $\mathbb{B}, \$V$ **in** E_1 **where** C_1 **and** C_2 **and** C_3 **groupby** $[\vec{B}]$
return E_3

(* If all variables in $\vec{\mathbb{B}}$ are bound to sequences that have no duplicates by id. $\vec{\mathbb{B}}$ is the list of variables defined in $\mathbb{B}$ that are visible to E_2. Practically the conditions are checked by verifying whether the expressions that variables in $\vec{\mathbb{B}}$ bound to are one of the following types: path expressions that include a"/" or"//" operator or an axis step; union, intersect, and except expressions. *)

(G7)

for $\mathbb{B}_1$,
  $\$V$ **in** (**for** $\mathbb{B}_2$ $(\textbf{where}\ C_1)?$ $(\textbf{groupby}\ G_1)?$
    return E_1),
  $\mathbb{B}_3$
$(\textbf{where}\ C_2)?$ **groupby** G_2 **return** E_2

$\mapsto$

for $\mathbb{B}_1, \mathbb{B}_2, \$V$ **in** $E_1', \mathbb{B}_3$ **where** C_1 **and** C_2 **groupby** G_2
return E_2

(* if G_2 contains only grouping by value expressions. $E_1' = E_1$ when G_1 is absent; otherwise $E_1' = \theta_{\$X_i \mapsto exp_i}(E_1)$ for every "$expr_i$ *as* $\$X_i$" in G_1. *)

(G8)

for $\$V$ **in** $\$V'$**groupby** G **return** E'

$\mapsto$

$\theta_{\$V \mapsto \$V'}(E')$

(* if $V' is not defined by **let** . $E' = \theta_{\$X_i \mapsto exp_i}(E)$ for every "$expr_i$ *as* $\$X_i$" in G. *)

(G9)

for $\mathbb{B}$ **groupby** G_1
return for X **in** X'**where** C **groupby** G_2
return E_r
$\mapsto \theta_{X \mapsto X'}$(**for** $\mathbb{B}$ **where** C **groupby** G_1 **return** E_r)

(G10)

for $\mathbb{B}_1, \$V$ **in** $V', \mathbb{B}_2$ **groupby** G **return** E
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}'_2$ **groupby** G' **return** E')(* if $V' is not defined by **let** *)
$\mathbb{B}'_2 = \theta_{\$V \mapsto \$V'}(\mathbb{B}_2)$
$G' = \theta_{\$V \mapsto \$V'}(G)$
$E' = \theta_{\$V \mapsto \$V'}(E)$

(G11)

for $\mathbb{B}_1$(**where** C_1)? **groupby** G_1 **return**
for $\mathbb{B}_2$(**where** C_2)? **groupby** G_2 **return** E
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}_2$ **where** C_1 **and** C_2 **groupby** G_1, G_2
return E
(* if G_1 and G_2 contain only grouping by value variables *)

(G12) **groupby** $\$V, \$V \mapsto$ **groupby** $\$V$

Notice that although normalization introduces grouping into the transformed queries, this does not involve any extra computational costs. The reason is the following. We introduce two types of grouping. Grouping by value is introduced by rules (G1), (G2), and (G3) in Table 2.5. These three rules just replace **distinct-values** functions with the computationally equivalent groupby-value clauses. groupby-id is introduced by rule (G4) for those variables bound to sequences where no duplicates are present. Hence, the underlying execution engine does not need to perform any grouping computation for groupby-id clauses, either.

Those FLWOR expressions for which one cannot introduce grouping clauses (for example, a groupby-id clause cannot be introduced for variables defined by functions which may return duplicate nodes) are the object of extensions presented in section 2.5. In appendix 2.A, we present a set of sound normalization rules that

apply to expressions going beyond pure NEXT.

2.4 The Rewriting Algorithm

Given a NEXT query Q and a set of NEXT views $\bar{V}$, algorithm NextRewrite equivalently rewrites the binding stage of Q to use solely the views. Since Q's tagging stage is independent of how the variable bindings were obtained, it is reused. NextRewrite obtains a query RW, such that (i) RW^{tag} is Q^{tag}, (ii) the binding stage of RW is expressed only in terms of the views and returns the same bindings as the binding stage of Q for each XML document D (preserving their order if ordered rewritings are sought): $Q^{bind}(D) = RW^{bind}(D)$. Since $RW = Q^{tag} \circ RW^{bind}$ and $Q = Q^{tag} \circ Q^{bind}$, it follows that RW is equivalent to Q.

To find RW^{bind}, NextRewrite proceeds in three phases.

Phase 1 identifies all variables x of Q whose bindings are also produced by some view V and can therefore be retrieved by navigation into V's output. This navigation is called an *alternate view access path* towards x's bindings. The view access paths are (redundantly) added to Q. Call the resulting expanded query Q_E.

Phase 2 restricts the expanded query Q_E by dropping all original query navigation and keeping only the view access paths added during the expansion. Query variables in the tagging and groupby components are appropriately replaced. Order-related checks are performed at this point when ordered rewritings are of interest. The result is the candidate rewriting RW, which performs a join of the alternate view access paths.

Finally, Phase 3 checks whether the candidate rewriting RW is truly equivalent to Q. Equivalence may fail if the join of the view access paths is lossy, i.e. returns too many bindings, or due to the loss of element identities caused by XQuery's copy semantics for result construction. This is not the case in our example.

2.4.1 Phase 1: Detecting View Access Paths

This phase detects alternate *access paths* through the views towards the bindings of the query variables. The access paths are detected via *mappings* from the views to the query and are implemented by navigation patterns which *invert the view return functions*. We detail these concepts next.

View access paths. Given a vector $\bar{x}_Q$ of Q's variables and an equal-arity vector $\bar{x}_V$ of V's variables, we say that there is an access path through $\bar{x}_V$ to $\bar{x}_Q$ iff (i) for all documents D, the bindings of $\bar{x}_Q$ are a subset of the bindings of $\bar{x}_V$, and (ii) the bindings of $\bar{x}_V$ are retrievable from the output of V.

Inverse of a return function. Views do not return their variable bindings directly, but instead construct new XML output from them. The view's return functions must hence be inverted to retrieve the bindings. Given the return function f of a NEXT block, we denote the *inverse* of f with $Inv(f)$. For each tuple of variable bindings t, $f(t)$ outputs an XML fragment x, while $Inv(f)(x)$ retrieves from x a copy of t. The reason only a *copy* of t (as opposed to t itself) is accessible is that XQuery semantics specifies that a return function's output is constructed by copying the XML subtrees to which the view variables are bound, thus losing the identities of element nodes [BCF+]. Phase 3 of the algorithm determines whether these copies suffice for rewriting. We illustrate the inversion below.

EXAMPLE 2.4.1. *In Example 2.1.1, there is an access path to Q's variables $\$r, \$a$ through V's variables $\$r, \$a$, whose bindings are retrievable by navigating into V's output. The navigation pattern of the access path is obtained from* $\mathrm{Inv}(f_1^V)$ *and* $\mathrm{Inv}(f_2^V)$ *(shown in Figure 2.9 (a)), by adorning with fresh variables each internal node of the tree corresponding to the return function's XML constructors. To capture the fact that the bindings of variable $\$a$ returned by f_2^V are children of the* `authors` *element constructed by f_1^V, we add a /-edge from $\$as$ to $\$a$. The resulting inverse functions specify that, in order to navigate to an* `author` *element in the view output, we need to first navigate to an* `authors` *element and then continue with a nested navigation to its children.*

Invertible NEXT views. In order to invert NEXT return functions, we sometimes need to go outside of the language and use navigation to certain positions

in a list. Take for instance the query

for $\$p$ **in** doc(in.xml), $\$x$ **in** $\$p/c, \$y$ **in** $\$p/c$,
return $\langle r\rangle\langle a\rangle\$x\langle/a\rangle\langle a\rangle\$y\langle/a\rangle\langle/r\rangle$

in which navigation to a children of the constructed r elements does not disambiguate between the bindings of $\$x$ and those of $\$y$. However, we can do so with an inverse function which navigates to the first a child for $\$x$, and the second for $\$y$. Notice that if in the above query, $\$y$ was instead bound to $\$p/d$, we could still disambiguate by navigating along $r/a/c$ for the bindings of $\$x$ and along $r/a/d$ for $\$y$.

The return functions of arbitrary XQueries are not necessarily invertible (consider aggregates for instance, where one cannot navigate into the aggregate result to reconstruct the arguments). However, a sufficient condition for a NEXT query to be invertible is that for each of its return functions f, (i) all nested **groupby** block arguments B of f appear within an element constructor in f: $\langle a\rangle B\langle/a\rangle$, or (ii) f has at most one nested **groupby** block argument.

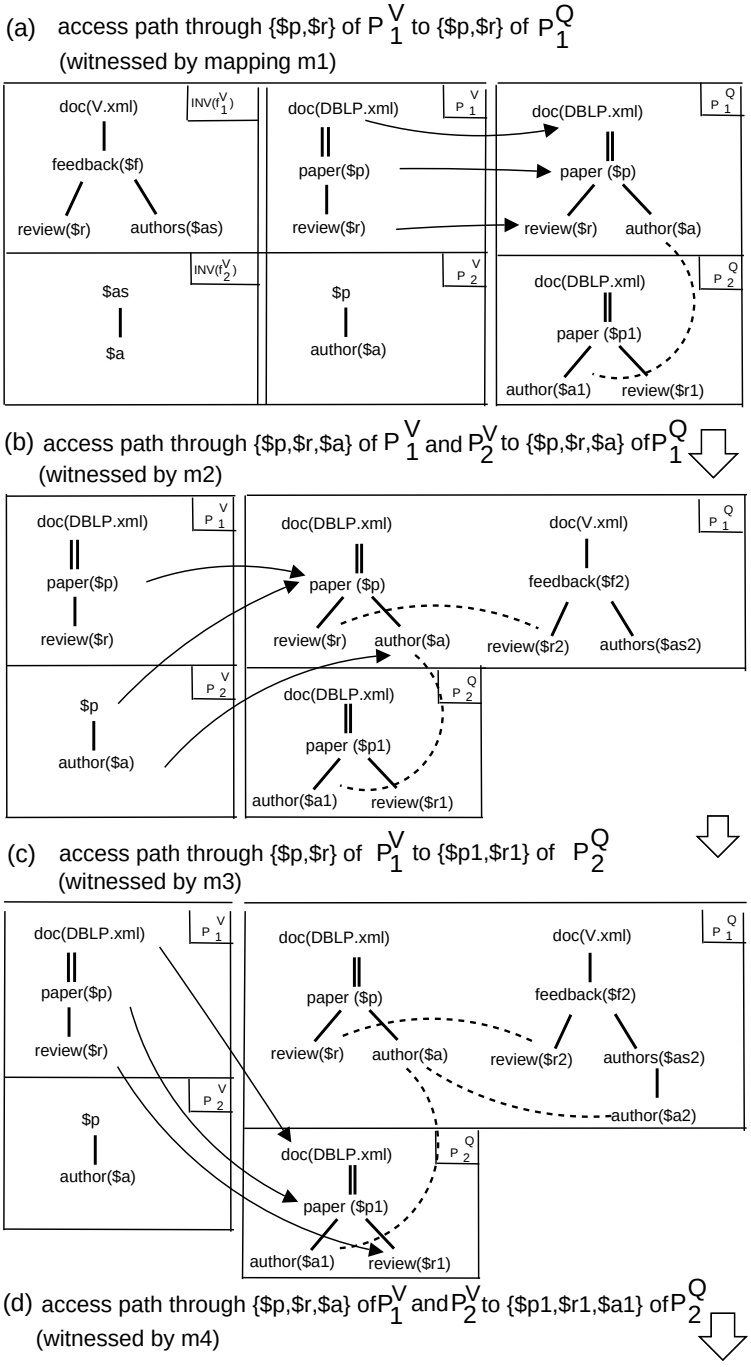


Figure 2.9: Phase 1 of NextRewrite for Q,V from Example 2.1.1 (continued on next page)

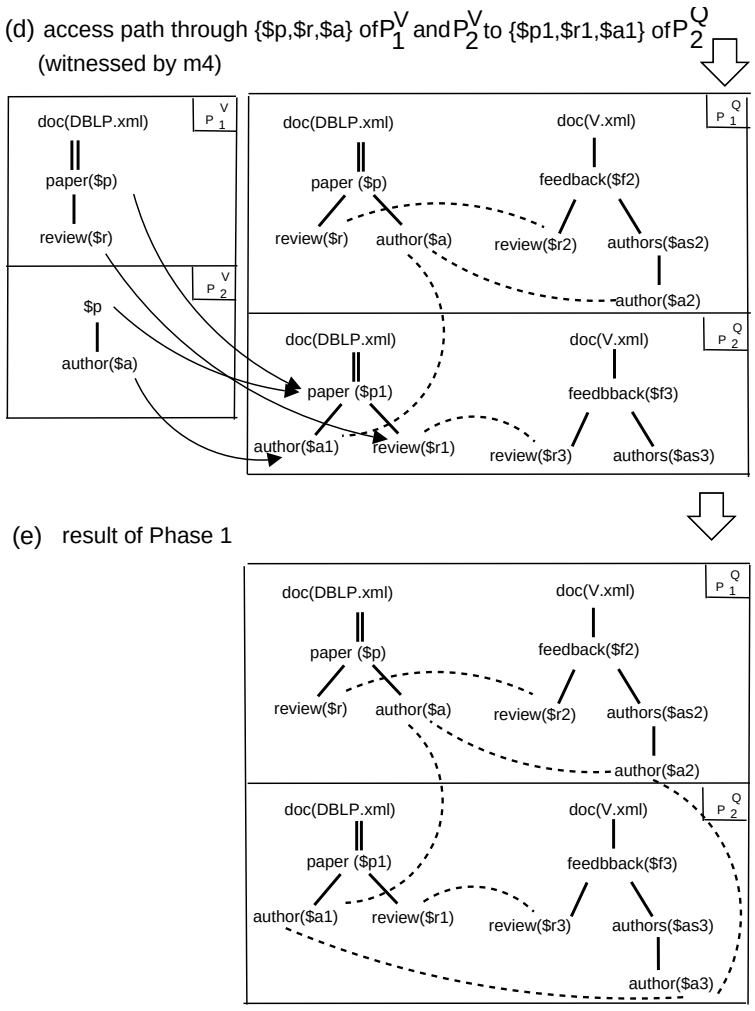


Figure 2.9: Phase 1 of NextRewrite for Q,V from Example 2.1.1 (continued from previous page)

Detecting access paths using mappings. We now focus on how access paths are detected. By definition, an access path through view variables $\bar{x}_V$ towards the query variables $\bar{x}_Q$ means that the set of bindings of $\bar{x}_Q$ is contained in the set of bindings of $\bar{x}_V$. We adopt the containment test from [DPX04], which characterizes containment by the existence of *NEXT pattern mappings*, which we shall henceforth call simply *mappings*. It follows from [DPX04] that there is an access path to $\bar{x}_Q$ through $\bar{x}_V$ if there is a mapping from V to Q which maps $\bar{x}_V$ into $\bar{x}_Q$, and if V's return functions are invertible.

Definition 2.4.1. *NEXT pattern mappings [DPX04] A* mapping *m from NEXT pattern P_1 to NEXT pattern P_2 maps the variables of P_1 into variables of P_2 such that*

(i) variables map to variables of the same tag label,

(ii) the source and target variables of each /-edge in P_1 map to the source, respectively target variable of some /-edge in P_2,

(iii) the source and target variables of each //-edge in P_1 map to the source, respectively target of a path of /- and //-edges in P_2,

(iv) for each id-equality v **is** *u in P_1, the equality $m(v)$* **is** *$m(u)$ is in the closure of P_2's id-equalities under reflexivity, symmetry, and transitivity, and*

(v) for each value-equality v **eq** *u in P_1, the equality $m(v)$* **eq** *$m(u)$ is in the closure of P_2's value-equalities under reflexivity, symmetry, transitivity, and the rule x* **is** *$y \Rightarrow x$* **eq** *y.*

EXAMPLE 2.4.2. *Figure 2.9 illustrates Phase 1 of algorithm* NextRewrite *on the running example. For every view access path to its variables, the query is expanded with the appropriate navigation given by the view inverses. Each snapshot shows the mapping (depicted by arrows) which detects an access path, and the successor snapshot shows the query after adding this access path. The value equality conditions (shown by dotted lines) record the correspondence between query variables and the view variables providing their alternate access path. Snapshots (a) and (b) show access paths detection for block B_1^Q, and similarly (c) and (d) for block B_2^Q. The result of Phase 1 is shown in Snapshot (e) which contains, besides the original query patterns, the NEXT pattern of the rewriting candidate RW (compare to Figure 2.6).*

Exploiting view block nesting for non-redundant mapping computation. According to XQuery semantics, when a nested block is correlated by shared variables to an ancestor block, the bindings of the nested block's free variables are provided by the ancestor block. This means that whenever a view block B^V provides an access path, the access to the bindings of B^V's free variables is provided by ancestor blocks of B^V. Therefore, the access path through B^V must be an extension of an access path through B^V's ancestors. Similarly, the mapping m witnessing

an access path through B^V must be an extension of a mapping m_p witnessing an access path through the ancestors. By extension, we mean that m and m_p agree on shared variables. To avoid redundant rediscovery of ancestor mappings, Phase 1 visits the view's **groupby** tree in a top-down fashion, finding mappings m_p from ancestor blocks once, and recursively extending them to nested blocks if possible.

EXAMPLE 2.4.3. *Snapshot (a) in Figure 2.9 shows the mapping* $m_1 = \{\$p \mapsto \$p, \$r \mapsto \$r\}$ *of* P_1^V *into* P_1^Q. *Mapping* m_2 *in Snapshot (b) is the extension of* m_1 *to* P_2^V *using* $\{\$a \in P_2^V \mapsto \$a \in P_1^Q\}$. *To avoid clutter, we only show the arrows for the extension.*

Congruence Closure. The congruence closure operation is needed to expose implicit mapping opportunities. See for instance Figure 2.10 for patterns P^V and P^Q. P^V has no mapping into P^Q, as the only way to map $\$u_2$ is into $\$x_2$, which has no d-child to map $\$u_4$ into. However, from the value-equality of $\$x_2$ with $\$x_4$, we can infer the existence of a d-child under $\$x_2$, call it $\$x_6$, which is value-equal to $\$x_5$ (and, symmetrically, that of a c-child $\$x_7$ which is value-equal to $\$x_3$). $\$x_6$ can now serve as target for $\$u_4$. Procedure CongruenceClosure (shown in the pseudocode below) makes this kind of inferences by exposing additional mapping targets.

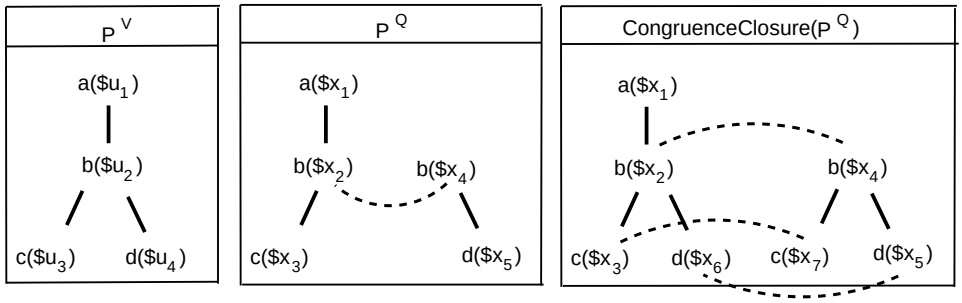


Figure 2.10: CongruenceClosure enables mappings

2.4.2 Phase 2: Candidate Rewriting

Phase 2 restricts the result of Phase 1, keeping only the view access paths. To this end, it identifies all variables that can be dropped from the query's group-by lists, substituting the remaining ones with view variables as dictated by the

view access patterns. If no appropriate view variables are found, the rewriting fails. Additional failure cases apply when ordered rewritings are sought, or due to the copy semantics of XQuery result construction (explained below). The new return functions of the rewriting are obtained by applying the same substitution to the query's return functions.

Dropping **groupby** *variables.* To check whether a **groupby** variable x can be dropped, we test that (i) x does not appear in any query return function, and (ii) in the **groupby** list, x appears together with some other variable y which determines x's value or id (depending on whether x is a groupby-value of groupby-id variable). This test is necessary to preserve the cardinality of the groups output by the binding stage. For groupby-id variables, the bindings of x are determined by those of y if y binds to children of x. [4]

Replacing **groupby** *variables.* When replacing a groupby variable v of the original query with a variable u of the view, we must make sure that the rewriting generates the same groups of variable bindings as the query. To this end, the bindings of both variables should ideally be identical (same value and identity). This means that v can only be replaced with variables u which are at least value-equal to v. In addition, if v is a groupby-value variable, it may be replaced with u regardless of whether u is a groupby-value or groupby-id variable, since the id of u does not contribute to the **groupby** result. But if v is a groupby-id variable, it must be replaced only by a groupby-id variable u since groupby-value variables lose the identity (and cardinality) of their bindings.

EXAMPLE 2.4.4. *Figure 2.6 displays the NEXT form of the candidate rewriting obtained for the query and view from Example 2.1.1 by restricting the Phase 1 result from Snapshot (e). The* **groupby** *tree block is isomorphic to that of Q and the patterns navigate exclusively into the views. The groupby-value variable* $\$a$ *has been replaced by* $\$a_2$*, since author values can be obtained from the view as well, as witnessed by the value-equality between* $\$a$ *and* $\$a_2$*. The groupby-id variable list* $\$p_1, \$r_1$ *has been first restricted to* $\$r_1$*, since it determines* $\$p_1$ *which is not used in*

[4] Other cases requiring XML Schema information are: (i) when x and y are siblings, both non-optional, sharing their tag names with no sibling; (ii) when y binds to x's parent, x shares its tag with no siblings, and is not optional; (iii) in the presence of an XML key constraint.

the return function and the nested blocks. Then $\$r_1$ *is substituted by* $\$r_3$ *according to the value-equality. All these replacements generate* f_1^{RW} *and* f_2^{RW} *in Figure 2.6, obtained from* f_1^Q *and* f_2^Q *in Figure 2.5 (by replacing variables* $\$a$ *with* $\$a_1$ *and* $\$r_1$ *with* $\$r_3$*).*

Issues of copy semantics. The copy semantics of XQuery views adds an additional technical problem: since the view's output elements are copies of input elements, they lose the original identities and there is no hope to find a view groupby-id variable u whose bindings have the same identities as those of v. Fortunately, it is sufficient if the identities of u's bindings are in one-to-one correspondence with those of v's bindings. This ensures the same number of groups regardless of whether we group by v or u, and the same outcome of **is** tests if we substitute v for u in them. If V's output extracted by u was created by copying the same elements as v binds to, then this one-to-one correspondence is guaranteed by the XQuery copy semantics. The above restrictions for groupby-variable replacement are only necessary, not sufficient for preserving the number of groups produced by the binding stage. The final check is performed in Phase 3.

2.4.3 Phase 3: Equivalence Check

Since the views may be under-conditioned, the candidate rewriting may contain the bindings of the query, but is not guaranteed to be equivalent. The equivalence check is similar to the relational case [LMSS95a]: First, unfold the views in the rewriting (for us, this means substituting for each inverse function the pattern corresponding to the same view block) so that the rewriting is expressed in terms of source documents. Next, check equivalence between Q and the unfolding of RW. The equivalence check was handled in detail in [DPX04], where it was employed in minimization of NEXT queries. Its key point is that equivalence of nested blocks has to be judged in the context of their ancestor blocks, since ancestor blocks provide the bindings for the variables which are free in descendant blocks.

EXAMPLE 2.4.5. *The unfolding of RW is shown in Figure 2.11.*

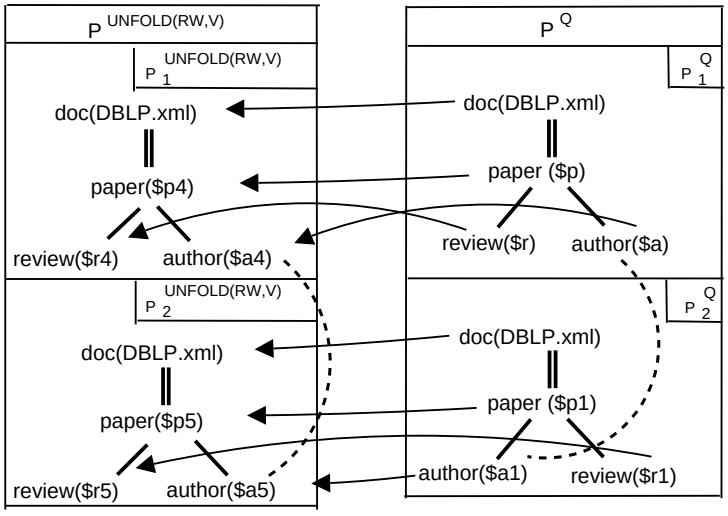


Figure 2.11: Sample mapping used in checking equivalence of Q with UNFOLD(RW, V)

2.4.4 Details and Formal Guarantees

The pseudocode of algorithm NEXTREWRITE is shown in Figures 2.12, 2.13 and 2.15. For a given query Q and view V, EXPANDALLBLOCKS visits Q's tree of **groupby** blocks in a top-down manner, invoking EXPANDBLOCK at each block B^Q.

Given a query block B^Q with pattern P^Q and a view block B^V with pattern P^V, EXPANDBLOCK searches for a subset of P^Q's variables to which P^V provides an alternate access path. They are detected via mappings (m'_p in line 4). Once a view access path is found, it is recorded by calling procedure ADDVIEWACCESS (line 5). The search continues recursively for access paths provided by blocks nested within B^V (lines 6–7). To avoid redundant computation of mappings from B^V during the visit of nested blocks, m'_p is passed as argument.

Detecting access paths within the query's ancestor context. Since a nested query block B^Q may have free variables correlating it with its ancestors, an access path may become possible only once we consider the extra constraints on the bindings of free variables, as provided by the patterns of B^Q's ancestors. Procedure EXPANDBLOCK takes this into account by mapping the view pattern P^V not just into P^Q, but into the *merged pattern* M^Q. M^Q is obtained by unioning together

NextRewrite($Q, \bar{V}$)

▷ *Phase 1: expand Q with alternate view access paths:*

1 **for** each $V \in \bar{V}$

2 **do let** B^Q, B^V be the root **groupby** blocks of Q, V

3 $X =$ ExpandAllBlocks(B^Q, B^V)

▷ *Phase 2: construct candidate rewriting using only view access:*

4 $RW =$ KeepViewsOnly($X, \bar{V}$)

▷ *Phase 3: check if candidate rewriting is equivalent to Q:*

5 **if** Unfold($RW, \bar{V}$) is not equivalent to Q

6 **then** report "no rewriting exists"

7 **else** minimize redundant access paths in RW and return result

Figure 2.12: Main Steps of Algorithm NextRewrite

the edges and equalities of P^Q with those of its ancestor patterns, and inferring any additional implied equalities according to procedure CongruenceClosure (lines 2–3).

Expanding P^Q with view access. Whenever P^V provides an access path to variables in P^Q, AddViewAccess expands P^Q with a copy of the navigation pattern corresponding to the inverse of the view return function, $Inv(f)$ (line 2). $Inv(f)$ may contain variables v which occur in P^V or in the ancestor patterns of P^V and are thus already mapped by m_p. These are the variables in domain$(m_p) \cap Inv(f)$ referred to in line 3. The image under m_p of each such v identifies a query variable $m_p(v)$ with an access path through v. This is recorded in line 4 by the value-equality between $m_p(v)$ and v_c, the access path variable introduced in line 2 to retrieve the v bindings for this access path.

EXAMPLE 2.4.6. *The mapping m_1 in Snapshot (a) witnesses the access path to $\$r \in P_1^Q$ through the inverse of f_1^V. Snapshot (b) shows the extension of P_1^Q with a copy of* Inv(f_1^V). *Line 3 of* AddViewAccess *computes* $\{\$r\} = domain(m_1) \cap$ Inv(f_1^V). *Line 4 adds the equality between the copy of variable $\$r$ (which is $\$r_2$) and $m_1(\$r)$ (which is $\$r \in P_1^Q$).*

ExpandAllBlocks(B^Q, B^V)

1 ExpandBlock($B^Q, B^V, m_\emptyset, m_\emptyset$) ▷ *$m_\emptyset$: empty mapping*
2 **for** each child B_c^Q of B^Q **do** ExpandAllBlocks(B_c^Q, B^V)

ExpandBlock(B^Q, B^V, m_p, m_r)

▷ *m_p: mapping of pattern vars. from B^V's ancestors into B^Q*
▷ *m_r: mapping of inverse function vars. from B^V's ancestors into access path vars in expansion of B^Q and ancestors*

1 **let** P^Q, P^V be the patterns of B^Q, B^V
2 **let** M^Q denote the pattern obtained by merging P^Q with the patterns of all ancestor blocks of B^Q
3 CongruenceClosure(M^Q)
4 **for** each mapping $m_p' : P^V \to M^Q$ which extends m_p
5 **do** $m_r' =$ AddViewAccess(P^Q, B^V, m_p', m_r)
6 **for** each child B_c^V of B^V **do** ExpandBlock(B^Q, B_c^V, m_p', m_r')
7 CongruenceClosure(P^Q)

AddViewAccess(P^Q, B^V, m_p, m_r)

▷ *adds to P^Q the inverse of B^V's return function*
1 **let** f be the return function of B^V, and *Inv*(f) its inverse (*Inv*(f) is a tree pattern)
2 add to P^Q a copy of the edges in *Inv*(f) (for each variable $v \in \text{vars}(Inv(f))$ denote its copy with v_c)
▷ *record equality between query vars. and their corresponding view access path vars.:*
3 **for** each $v \in \text{vars}(Inv(f)) \cap \text{domain}(m_p)$,
4 **do** add to P^Q the value-equality $m_p(v)$ **eq** v_c
▷ *link free variables in the copy of* Inv(f) *to*
▷ their bound occurrence in ancestor (from m_r):
5 **for** each $v \in \text{vars}(Inv(f)) \cap \text{domain}(m_r)$, **do** replace v_c with $m_r(v)$ in P^Q
6 construct extension m_r' of m_r that maps each $v \in \text{vars}(Inv(f)) \setminus \text{domain}(m_r)$ into v_c
7 **return** m_r'

Figure 2.13: Phase 1 of Algorithm NextRewrite

CongruenceClosure(P) ▷ *side-effects* P

1 extend P with the symmetric, transitive closure of its equalities
2 **for** each value-equality v **eq** $u \in P$ and each child v' of v
3 **do if** u has no child u' with v'**eq** u'
4 **then** add a copy of the subtree rooted at v' as a child of u
5 add a value-equality between each copied variable and its copy
6 analogous to lines 2–5 for id-equalities v **is** $u \in P$

Figure 2.14: Congruence Closure used in Algorithm NextRewrite

Correlating view access paths. The argument m_r of AddViewAccess is used to handle free root variables $\$v_a$ in $Inv(f)$. These variables also occur in the inverse function $Inv(f_a)$ of some ancestor block B_a^V of B^V, thus correlating $Inv(f)$ and $Inv(f_a)$. By the time AddViewAccess is invoked for B^V, it has already executed on B_a^V and has introduced a copy of $\$v_a$ into B_a^V. Upon reaching B^V, AddViewAccess must preserve the correlation by using the same copy of $\$v_a$. To this end, it consults the mapping m_r which records the correspondence between correlation variables in $Inv(f)$ and their copies in $Inv(f_a)$ (lines 5–6). m_r is then extended to m_r' (line 7) to record the new correspondences between correlation variables in $Inv(f)$ which appear in nested blocks of B^V (line 8).

EXAMPLE 2.4.7. *In Figure 2.9, Snapshot (a)* $m_r = m_\emptyset$*, and* $\mathrm{Inv}(f_1^V)$ *has no free variables. According to line 7 in procedure* AddViewAccess*,* $m_r' = \{\$f \mapsto \$f_2, \$r \mapsto \$r_2, \$as \mapsto \$as_2\}$ *is constructed to record the actual names of the variables introduced as copies of those in* $\mathrm{Inv}(f_1^V)$*. In Snapshot (b)* AddViewAccess *is called with* m_r'*.* AddViewAccess *now extends* P^Q *with a copy of* $\mathrm{Inv}(f_2^V)$*, yielding the pattern from Snapshot (c). Notice that this copy is added below the* $\$as_2$ *variable due to lines 5–6 in* AddViewAccess*;* m_r' *has recorded that* $\$as_2$ *is the copy of the occurrence of* $\$as$ *in* $\mathrm{Inv}(f_1^V)$*, and the same copy is consistently used for the occurrence of* $\$as$ *in* $\mathrm{Inv}(f_2^V)$*.*

KeepViewsOnly($X, \bar{V}$)

1 **for** each block B^X in query X, let P^X be its pattern
2 Replace P^X with the restricted pattern P^{RW} which keeps from P^X only variables reachable along paths of /- and //- edges from roots mentioning view names. Keep the edges and equalities involving them.
3 Drop from B^X's **groupby** list all variables which are:
– uniquely determined by other variables in the list (due to parent-child relationship), and
– do not appear free in nested blocks
4 Replace the remaining variables v in B^X's **groupby** list with variables $u \in P^{RW}$ s.t.:
5 v **eq** $u \in P^X$;
6 if v is a groupby-id variable, u was introduced during Phase 1 as the correspondent of a view's groupby-id variable;
7 if v is a groupby-value variable, u was introduced due to either a view's groupby-value or groupby-id variable;
8 **if** not all groupby variables can be replaced, **then** report "no rewriting found"
9 **if** an ordered rewriting is sought and no ordering of the view pattern roots preserves the pre-replacement ordering of **groupby** variables
then report "no ordered rewriting found"

Unfold($RW, \bar{V}$)

1 **let** B^{RW} be the root **groupby** block of RW
2 **return** UnfoldBlock($B^{RW}, \bar{V}$)

UnfoldBlock($B^{RW}, \bar{V}$)

1 **for** each view access path in B^{RW} introduced based on the mapping m defined on the view pattern P^V
do add to B^{RW} a copy P^U of P^V (replacing all variables with fresh variables)
record value equalities between variables from P^U and the corresponding variables in the initial B^{RW}
4 CongruenceClosure(B^{RW})
5 **for** each child B^{RW}_C of B^{RW}
6 **do** UnfoldBlock($B^{RW}_C, \bar{V}$)

Figure 2.15: Phase 2 of Algorithm NextRewrite

Theorem 2.4.1. *Let the input to* NextRewrite *be a NEXT query Q and a set $\bar{V}$ of invertible NEXT views.*

1. (Soundness) *The NEXT query* RW *output by* NextRewrite *(if any) is an equivalent rewriting of Q.*

2. (Unordered Completeness) *If Q's binding stage Q^{bind} has some equivalent unordered NEXT rewriting using only $\bar{V}$, then* NextRewrite *is guaranteed to find an equivalent rewriting* RW *of Q using only $\bar{V}$.*

Lemma 2.4.1. *$Q \equiv Q_{ext}$, where Q_{ext} is the result of Phase 1.*

Proof:

For a mapping $m : P^V \rightarrow P^Q$, the bindings for the nodes $\bar{x}$ from $m(B^V)$ can be alternatively obtained by running the pattern of the view, B^V. This can be proven by induction over the nested trees of patterns. Mappings for the base case are similar to mappings between simple tree patterns. Mappings for descendant **groupby** blocks are computed in ExpandBlock by merging the pattern with the pattern of the ancestor blocks. (Thus, we can re-use techniques based on pattern mappings that led to sound and complete algorithms for rewriting using views and containment of XPath [XÖ05] [MS04].)

Hence, the variables appearing in the return function of the view are a subset $\bar{u} \subseteq m^{-1}(\bar{x})$ and their bindings include the bindings for the corresponding subset of $\bar{x}$.

The bindings for $\bar{u}$ can be reached by inverting the return function $f(\bar{u})$ of the view and thus building the new access paths from Q_{ext}. Adding the pattern of $Inv(f)$ does not change the semantics of the query, as all conditions checked by the navigation of $Inv(f)$ are already fulfilled by the navigation of the query pattern. Neither do the inferred equalities between query and view access paths variables, as the bindings of the latter include the bindings of the former. Thus, at the end of Phase 1 we obtain a query extension which is equivalent to the original query, but contains additional navigation into the output of the views.

■

Lemma 2.4.2. RW^{bind} *(resulting after Phase 2) outputs a superset of the bindings of* Q*.*

Proof: This is a consequence of the fact that the view access paths provide a superset of the bindings for the query access paths and that all output (groupby) nodes of the query can be replaced by nodes from view access paths. Restrictions on the patterns of view access paths, imposed after CongruenceClosure, must also be taken into account. But they only add equalities and navigation conditions that were already true on the corresponding fragments from the pattern of Q, because $Q \equiv Q_{ext}$ (lemma 2.4.1). Therefore the containment still holds. ■

Proof: [Proof of Theorem 2.4.1]

Since the return functions are the same up to a renaming of variables, we only need to prove that the binding stage outputs the same tables of variable bindings for both Q and RW.

Phase 3 verifies that $RW^{bind}(D) \equiv Q^{bind}(D)$ for any instance D, that is it checks the soundness of the rewriting. It cannot do it directly, as RW is written on the view schema, that is why UNFOLD is called to obtain $U = \text{UNFOLD}(RW, \bar{V})$. U is just an extension of RW with the navigation from the view bodies corresponding to the mappings used when constructing RW. Hence, this additional navigation does not impose any additional constraint, as the bindings for variables in RW were obtained by running these fragments of U. Thus, we have that $U \equiv RW$, and checking that $Q \equiv U$ is equivalent to checking that $Q \equiv RW$.

For the completeness part, suppose that Q has some equivalent rewriting R' using only $\bar{V}$.

By theorem 2.4.2, NextRewrite computes all the mappings view-query, while building the candidate rewriting RW. Suppose now that NextRewrite fails with the message "no rewriting found", implying that there is at least one **groupby** variable y of Q whose variables cannot be retrieved using any view access paths. This gives a contradiction with the existence of R', because there is an access path for y in R', obtained using $\bar{V}$.

If RW is indeed a rewriting, from lemma 2.4.2, $R' \equiv Q \subseteq RW$, hence R' is also contained in RW. Both RW and R' provide access paths towards the bindings

of Q's **groupby** variables and the bindings produced by R' are subsumed by those produced by RW. But since RW has only child navigation, it follows that the access paths of R' are equivalent to a subset of the paths of RW. On the other hand, all equality conditions in RW were imposed starting from equality conditions from Q, and these conditions are also reflected by the variables of R', because $Q \equiv R'$. Hence RW is more restrictive than R' and $RW \subseteq R'$. We can thus conclude that $RW \equiv R'$ if such an R' exists.

■

It should not be surprising that we cannot guarantee completeness for ordered rewritings: in this case, even the equivalence of ordered NEXT queries is undecidable (this is a corollary of results in [Van05]). Our approach performs a best effort in that case, remaining sound (the equivalence check is now only sufficient, not necessary) and in practice still finding ordered rewritings in many cases.

2.4.5 Finding Mappings Efficiently

We specify next how the algorithm NextRewriting finds mappings. This is the most crucial step for the performance of the algorithm since (i) it is expensive (indeed, it is the only step that is not of polynomial-time complexity) and (ii) is invoked repeatedly.

We first present the prior work on finding mappings. For relational conjunctive queries, checking the existence of a mapping from V to Q is NP-complete in the number of variables of V [CM77]. The NP-hardness lower bound transfers to finding mappings from a NEXT pattern P^V into a NEXT pattern P^Q.[5] However, checking containment mappings between simple tree patterns without value equalities used in prior XPath works is performed in polynomial time in the size of the source pattern [AYCLS02, Ram02, MS04, GKP03]. Notice that the differences between simple tree patterns and NEXT patterns are (i) the number of distinguished variables, i.e., variables that are projected in the output (tree patterns

[5]Sketch of Proof: Given a relational mapping problem, assume that the relations are encoded in XML using, for instance, the default encoding of [CKS+00]. Then encode the relational conjunctive queries as single-block (i.e. non-nested) NEXT queries over the default encoding. It follows that finding NEXT mappings is as hard as finding conjunctive query mappings.

have only one, NEXT patterns have as many as there are groupby variables) and (ii) the number of variable equalities (tree patterns have none). It is therefore natural to seek an algorithm that runs in PTIME on tree patterns (no equalities, one distinguished variable) and whose performance degrades only with increasing number of groupby variables and/or equalities.

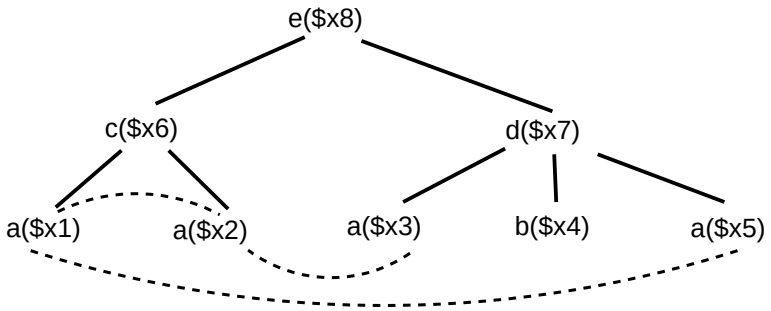


Figure 2.16: A NEXT Pattern ($\$x_5$ is the groupby-value variable)

We start by analyzing the complexity of computing mappings from a pattern P. For each (node and corresponding) variable $v \in P$, define $gVars(v)$ to be the set of grouping variables among the proper descendants of v in P. For the example pattern in Figure 2.16, assuming that $\$x_5$ is the only groupby-value variable, $gVars(\$x7) = \{\$x5\}$. Also, let $e(v)$ be all of v's proper descendants in P which are involved (directly or transitively) in equalities with variables who are not v's descendants. For instance, $e(\$x7) = \{\$x3, \$x5\}$. The symmetric, transitive closure of the equality conditions in P partitions its variables into *equality equivalence classes*. Given a set of variables $\mathcal{V}$ denote with $\#eqCls(\mathcal{V})$ the number of distinct equality equivalence classes represented in $\mathcal{V}$. For instance, $\#eqCls(\{\$x7, \$x3, \$x5\}) = 2$, since $\$x3$ and $\$x5$ are in the same equivalence class.

Define the *width* of v as

$$width(v) = \begin{cases} \#eqCls(gVars(v)), & \text{if } v \text{ is root} \\ \#eqCls(\{v\} \cup gVars(v) \cup e(v)), & \text{otherwise} \end{cases}$$

and define $width(P)$ as the maximum width of a variable in P. For example, $width(\$x7) = 2$ and $width(\$x8) = \#eqCls(\{\$x5\}) = 1$.

Note: *width*(P) is related to the *tree width* [CR97, FFG01] of P when regarded as a graph $\mathcal{G}_P$ whose edges consist of the /-, //-edges and the closure of equalities. We can obtain a tree decomposition of $\mathcal{G}_P$ by considering the navigation tree $\mathcal{N}$ of /- and //-edges and by propagating **groupby** variables up to the root and variables in the same equivalence class up to the lowest common ancestor in $\mathcal{N}$. At each node, only one node from the same equivalence class is kept. We can prove that the width of this decomposition is exactly our notion of *width* of a variable.

For queries with bounded tree width, previous work [CR97, FFG01] presents polynomial-time evaluation algorithms corresponding to the generalization of Yannakakis' algorithm [AHV95] from acyclic queries to bounded-tree-width queries. Our algorithm for computing mappings adapts these ideas. It is based on two key ideas. First, instead of computing mappings from P^V to P^Q one-at-a-time, we take a set-at-a-time approach by interpreting P^V as a query evaluated over the tree structure of P^Q.[6] P^V is compiled into a query execution plan in which we adapt from Yannakakis' algorithm [AHV95] the idea of pushing projections before the join in the bottom-up evaluation of the plan. Our experiments show that the plan compilation and subsequent optimization require negligible time. In addition, this time is not critical as these tasks are performed off-line, before query arrival.

The operators used in the mapping computation plan output flat relational tables. We use relational join, selection, and projection operators, as well as two special operators that are evaluated over the structure of P^Q. First, $\mathsf{scan}_{l \mapsto v}()$ returns the nodes in P^Q whose label is l. The result is a set of unary tuples whose attribute is called v. Second, $\mathsf{parents}_{l,c \mapsto p}(R)$ takes as input a set of tuples R and extends each of them with a new attribute p. Given a tuple r with $r.c = n$, where n is a node of P^Q, the new value $r.p$ is the l-labeled parent n' of n, assuming n' and n are connected via an /-edge in P^Q. If no such parent exists, the tuple is dropped. Similarly for $\mathsf{ancestors}_{l,d \mapsto a}$, d is the attribute holding the descendant and a is the extension attribute holding an ancestor with label l.

[6] This perspective stems from relational theory, in which mappings (homomorphisms) from a query V to a query Q are computed by regarding Q as a symbolic database and evaluating V over it [CM77].

The plan for a pattern P is obtained by using scan operators for the leaves of P, and join operators for its internal nodes. Moreover, for each operator o corresponding to a pattern variable v, add on top of o a projection operator whose projection list contains precisely the variables contributing to $width(v)$.

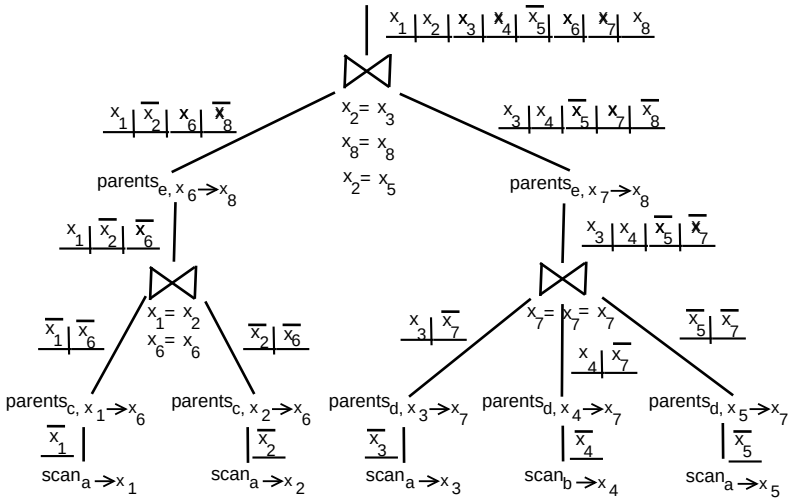


Figure 2.17: Algebraic plan for pattern in Figure 2.16

EXAMPLE 2.4.8. *We illustrate the translation into a plan on the pattern P from Figure 2.16. Given a pattern P^Q, the plan shown in Figure 2.17 finds all images of $\$x_5$ under mappings from P into P^Q. Each plan operator o is adorned with a schema S, whose overlined attributes A specify the projection list. The invariant maintained by the plan is that the sub-plan rooted at o computes the images of the A variables under all partial mappings of the S variables into P^Q. For instance, the topmost join operator finds the images of x_5 under the mappings of all eight variables of P.*

The join operators enforce the equality conditions, as well as the structural constraint requiring two variables to have the same parent variable. A condition $x = y$ evaluates to true on two tuples t_l (from the left join operand) and t_r (from the right join operand) if $t_l.x$ is equal to $t_r.y$ in P^Q; equality may be transitively implied by P^Q's equality conditions. The reader can check that in Figure 2.17, the number of overlined attributes for each operator o corresponds to the width of v.

Theorem 2.4.2. *Let P^V, P^Q be NEXT patterns. Then the evaluation over P^Q of the plan for P^V produces the images of the output nodes of P^V for all the mappings $m : P^V \rightarrow P^Q$ and runs in time polynomial in the size of P^Q and exponential in $width(P^V)$.*

Proof: The evaluation plan for a pattern P corresponds to the efficient evaluation of a tree decomposition of the graph $\mathcal{G}_P$ defined above. To each node n, corresponding to variable v, from P^V we associate a table $t(n)$ having *width(v)* fields from the projection list. Each internal node joins its children and projects the result on the variables from its table.

The completeness part can be proven by induction on the structure of the query plan. The leaf/scan nodes can map to any nodes in P^Q having the same label. In a bottom-up evaluation, an internal node n computes, for the subtree P^V_n rooted at n, the images of the variables in its projection list for all mappings of P^V_n that satisfy equality conditions involving only P^V_n. From definition 2.4.1 it follows that to preserve the images for the current node, the output nodes and nodes involved in equalities, it is enough to check at an upper level, by joins, that the resulting tuples correspond to projections of valid mappings.

The join and the projection associated to the node n corresponding to a variable v run in PTIME in the size I_n of its input and outputs $\mathrm{O}(I_n^{width(v)}) \leq \mathrm{O}(I_n^w)$ tuples, where $w = width(P^V)$. We start from the leaves, which are scan operators, running in $\mathrm{O}(|P^Q|)$ time. By induction, each step is polynomial in $|P^Q|^w$ and there are only $|P^V|$ such steps. ■

Corollary 2.4.1. *For any fixed $k \geq 1$ such that $width(P^V) \leq k$, checking the existence of a mapping $m : P^V \rightarrow P^Q$ can be done in polynomial time.*

Since pure tree patterns contain no equalities and only one distinguished variable (corresponding in NEXT to one groupby-id variable), simple tree patterns have width 1, and the result on polynomial-time computability of tree pattern mappings [AYCLS02] follows as a consequence of Corollary 2.4.1.

2.5 Beyond NEXT Rewriting

We now extend the applicability of the NEXT rewriting algorithm to arbitrary XQueries. This involves (i) extending the NEXT notation to accommodate arbitrary XQueries and (ii) extending the rewriting algorithm accordingly. We sketch the required extensions in this section. We preserve the approach used for NEXT rewriting: represent queries using patterns and find view access paths by matching the view patterns against the query patterns.

This extension leads to an algorithm which may not be complete: for queries going beyond NEXT, it might not find a rewriting, even if there is one. We show how its applicability can be improved by incorporating more information about the meaning of the XQuery expressions, captured as function properties.

We also prove below that we cannot hope for a complete algorithm, as the problem in general is undecidable.

For that we first introduce the problem of *emptiness*. Deciding the emptiness of a query means deciding whether the query returns the empty sequence for all possible input instances. Vansummeren [Van05] gives a small fragment of XQuery (only seven constructs) for which emptiness is already undecidable. Van den Bussche, Van Gucht and Vansummeren [dBGV05] present another XQuery fragment, with an unordered semantics, for which equivalence, and hence emptiness, is undecidable.

Theorem 2.5.1. *Let $\mathcal{X}$ be a sublanguage of XQuery for which emptiness is undecidable. Let $\mathcal{X}'$ be a sublanguage of XQuery that includes $\mathcal{X}$ and also downward navigation and predicates. Then the existence of an XQuery rewriting for an XPath query using views from $\mathcal{X}$ is also undecidable.*

Proof: Let Q_1 be a query from $\mathcal{X}$ and let $\$doc$ be a variable bound to the document node of the input instance. Adapting an idea from [SV05], we build the view V and the query Q below, where *step* is an element name that is not mentioned in Q_1:

$$V: \ \$doc[Q_1]//step$$

$$Q: \ \$doc//step$$

Q has an XQuery rewriting using V if and only if Q_1 never returns empty. Hence if the complement of emptiness is undecidable for $\mathcal{X}$, then rewriting queries using views is also undecidable for the languages defined in the statement of the theorem. ■

Observation The test of emptiness in the view language can be equivalently replaced by logical quantifiers.

Corollary 2.5.1. *The existence of an XQuery rewriting for an XQuery query using XQuery views is undecidable, both under set and list semantics.*

When choosing $\mathcal{X}$ to be the XQuery fragment from [Van05] or from [dBGV05], we obtain again undecidability for the existence of a rewriting.

2.5.1 Treating the Full XQuery

Extending NEXT: NEXT+. The solution we adopt for (i) is to abstract the non-NEXT subexpressions as *uninterpreted functions*, i.e. functions about whose semantics we make no assumptions. This allows us to uniformly capture all XQuery primitives which are ruled out by the OptXQuery syntax: aggregate functions, built-in predicates other than equality, universal quantification, negation, disjunction, user-defined functions, etc. Each of these are treated as some function F whose arguments are in turn NEXT expressions extended with uninterpreted function calls. We call this notation *NEXT+*. Its syntax corresponds to extending the NEXT grammar in Figure 2.4 with the productions

$$
\begin{array}{rcl}
FC & ::= & F(XQ_1, \ldots, XQ_l) \\
XQ & ::= & \textsf{for } Var_1 \textsf{in } (FC_1 \mid Path_1), \ldots, Var_n \textsf{in } (FC_n \mid Path_n) \\
 & & (\textsf{where } CList)? \\
 & & \textsf{groupby } (FC'_1 | [FC'_1] | \{FC'_1\}), \ldots, (FC'_k | [FC'_k] | \{FC'_k\}) \\
 & & \textsf{return } FC'' \\
Cond & ::= & FC \text{ - - where the function } F \text{ is (coercible to) boolean}
\end{array}
$$

In the syntax of groupby clauses above, $\{FC'_j\}$ indicates grouping by the *position* in the sequence of bindings. This new type of distinguished expressions was needed in order to handle sequences of bindings that may contain duplicates, as well as values that have no node ids. NEXT+ contains NEXT as a particular case when only paths appear in the **in** clauses, $FC'_1, \ldots, FC'_k$ are calls of the identity function with variables as arguments, there is no grouping by position, and the function in FC'' consists only of element construction and concatenation. The NEXT+ notation features additional nested block occurrences, where blocks are now not only circumscribed by **groupby** blocks, but also by the arguments of function calls. For instance, in the above production for XQ, the arguments of $FC_1, \ldots, FC_n, FC'_1, \ldots, FC'_k, FC''$ are function blocks nested within the block given by the outer **for** –**where** –**groupby** –**return** expression. Just like NEXT blocks, nested NEXT+ blocks may have free variables bound in their ancestor blocks. For example, consider the following query Q' which returns the reviews of multi-author conference papers that are available on the Web:

for $\$p$ **in** *web-available*($\$doc$), $\$r$ **in** $\$p/review$
where *count*($\$p[conference]/author$) > 1 **return** $\$r$

where the *web-available*() function may be implemented by concatenating the list of papers that have an electronic edition (*ee*) citation page with those appearing in a table of contents HTML page (*url*), as in the expression: *$doc//paper[ee], $doc//paper[url].* However, we suppose that the implementation of the function is not available, hence it cannot be inlined.

Normalization into NEXT+ yields three blocks B_1, B_2, B_3:

$$B_1 \left\{ \begin{array}{l} \textbf{for } \$p \textbf{ in } \textit{web-available}(\underbrace{\$doc}_{B_2}), \$r \textbf{ in } \$p/review \\ \textbf{where } F \underbrace{\left(\begin{array}{l} \textbf{for } \$a \textbf{ in } \$p/author \\ \textbf{for } \$c \textbf{ in } \$p/conference \\ \textbf{groupby } [\$a] \\ \textbf{return } \$a \end{array} \right)}_{B_3} \\ \textbf{groupby } \{\$p\}, [\$r] \\ \textbf{return } \$r \end{array} \right.$$

where F is the boolean function $\lambda N.\ count(N) > 1$ (in lambda notation). Note that *web-available*() may return duplicates, as the set of papers having an electronic edition is not necessarily disjoint from the the set of papers appearing in tables of contents.

In order to support duplicates and grouping-by-position, we need to modify accordingly the binding stage presented in Section 2.2 and Table 2.1. The block attributes in the nested binding tables may in general also contain bags and not only sets of bindings, as it was the case for pure NEXT. Besides group-by variables, the table may also contain *hidden* variables, bound in NEXT+ for- or some-clauses but not appearing in the groupby list.

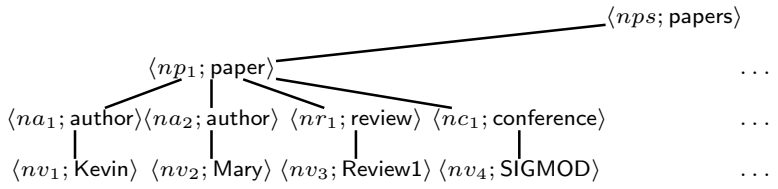


Figure 2.18: Input XML data for the NEXT+ query Q'

Figure 2.18 shows a modified version of the data from Figure 2.1 in which the first *paper* element also has a *conference* subelement. Suppose that the call to *web-available* in Q' returns np_1 followed by a duplicate of np_1. Table 2.6 shows the result of the binding stage. The table for block B_1 from Q' contains all bindings for the groupby-position $\$p$ variable, duplicates allowed, and for each such binding, all the distinct bindings for $\$r$. The sub-table for B_3 includes bindings for the groupby-value variable $\$a$, but also for the hidden variable $\$c$.

Hidden variables do not contribute to the output of a block, but they must be taken into account in order to check that the conditions tested by the corresponding clauses hold. Hidden variables do not change the cardinality of the output, as their bindings are nested after all groupby-variables.

When grouping only by value, the semantics was similar to unordered relational algebra with nesting. However, the other forms of grouping, when used

Table 2.6: Result of binding stage for NEXT+ blocks B_1 and B_3

B_1			
$\{\$p\}$	$[\$r]$	B_3	
np_1	nr_1	$[\$a]$	$\$c$
		nv_1	nc_1
		nv_2	nc_1
np_1	nr_1	$[\$a]$	$\$c$
		nv_1	nc_1
		nv_2	nc_1

in queries with ordered semantics, require that the output sequence respect a certain order: groupby-id sorts in document order and groupby-position sorts by the position in the original sequence.

Grouping by position (and by id also) could be implemented as in [RSF06], by introducing new indices for all ordered sequences. In this case, we can keep using regular sets instead of bags, using the index for disambiguation.

Rewriting NEXT+ Queries Our goals are similar as in the case of pure NEXT: given a NEXT+ query Q and a set of NEXT+ views $\mathcal{V}$, find a rewriting R of the Q using $\mathcal{V}$ such that the output of R and Q is the same for a given input XML document. This is an undecidable problem, even for a restriction of NEXT+ to NEXT queries with function calls, as these can perform arbitrary complex computations. However, we will restrict ourselves to a weaker notion of equivalence, in which all functions, including the ones we introduce for non-NEXT XQuery constructs, are uninterpreted. In this case, to obtain an equivalent rewriting over all XML instances, function calls have to be matched by exactly the same function names, only the arguments may differ. It is likely that we could guarantee completeness under this weaker equivalence, based on the completeness of our algorithm for NEXT queries.

Increased Complexity Another consequence is that we have to open the rewriting procedure to be able to deal with free variables, bound in upper blocks,

as in the example above, in which the variable $\$p$ defined in B_1 is used in B_2. This extension is not trivial, because, as shown in [DHT04b], verifying equivalence and containment of XML queries in general may also necessitate comparing tagging stages that build the output and not only the navigational part, as for NEXT rewritings using views. And even if we succeed in proving that it is enough to look at the navigational stages, the increased complexity of the language leads to an increased worst case time complexity of the rewriting problem and an increased space complexity of the output. That is why optimizations such as the ones described in section 2.7 can have a significant impact. In addition, by going to XQuery, we need to guarantee soundness in the presence of list semantics or a mixture of list and set semantics, order-by clauses and other sorts of expressions that affect order.

Extending NextRewrite. Since algorithm NextRewrite operates on nested blocks regardless of how these were created during normalization, only minimal changes are required. We still use mappings to detect access paths, still considering ancestor blocks to provide the context for the free variables. However, we must extend mappings to account for function calls. Our treating functions as uninterpreted requires extending the definition of mappings in [ODPC06] such that function calls $F(a_1, \ldots, a_n)$ match only against calls of the same function $F(b_1, \ldots, b_n)$, and only if, recursively, a_i is equivalent to b_i for each i.

Suppose there exists a view W, a modified version of the view from example 2.1.1, which only iterates over the web-available papers:

```
let $doc := document('DBLP.xml')
for  $p in web-available($doc), $r in $p/review
return  ⟨feedback⟩{ $r,                                                    (W)
                ⟨authors⟩{ $p/author }⟨/authors⟩
        }⟨/feedback⟩
```

with the NEXT+ form

$$
B_1^W \left\{ \begin{array}{l}
\textbf{for}\ \$p\ \textbf{in}\ \ \mathit{web\text{-}available}(\underbrace{\textbf{document}(\text{`}\mathit{DBLP.xml}\text{'})}_{B_2{}^W}), \\
\qquad \$r\ \textbf{in}\ \$p/\mathit{review} \\
\textbf{groupby}\ \{\$p\}, [\$r] \\
\textbf{return}\ \ \langle \mathsf{feedback} \rangle \{\ \$r, \\
\qquad \langle \mathsf{authors} \rangle \{ \\
\qquad\quad B_3^W \left\{ \begin{array}{ll}
\textbf{for}\ \$a\ \textbf{in}\ \$p/\mathit{author} & (NEXT_W) \\
\textbf{groupby}\ [\$a] & \\
\textbf{return}\ \$a &
\end{array} \right. \\
\qquad\quad \}\langle /\mathsf{authors} \rangle \\
\qquad \}\langle /\mathsf{feedback} \rangle
\end{array} \right.
$$

In our example, the algorithm generates the following candidate rewriting using the view W:

for $\$f$ **in document** $(\text{`}W\text{'})/\mathit{feedback}, \$r$ **in** $\$f/\mathit{review}$
where $\mathit{count}(\$f/\mathit{authors}/\mathit{author}) > 1$ **groupby** $[\$r]$ **return** $\$r$

Clearly, the more of the query is abstracted to functions, the less mappings will be discovered, resulting in fewer rewriting opportunities. We take particular care to maximize these opportunities by abstracting only the truly non-NEXT query primitives.

The advantage of our approach is twofold. First, we do not reject non-NEXT queries flat out, performing a best rewriting effort instead. Our approach combines the benefits of complete rewriting feasible for NEXT and the easy-to-engineer but incomplete techniques based on isomorphically matching common sub-expressions between query and view [DFK$^+$04]. Second we enable an extensible rewriting module, described in Section 2.5.2, which can be incrementally enhanced by adding partial information about the uninterpreted functions (thus interpreting them partially).

We also allow for function calls not to be covered by views, as long as the blocks inside the arguments of these calls are covered. Consider a modified version of the query from the beginning of this section that returns the entire content of

web available documents with more than one author:

for $*doc* **in document** (“*collection*”)/*docs*/*doc*

for $*p* **in** *web-available*($*doc*)

where *count*($*p*/*author*) > 1

return $*p*

We can rewrite this query with the view **document** (“*collection*”)/*docs*/*doc* that is used to provide bindings for the $*doc* variable and letting the query perform the call to the *web-available* function.

This type of rewritings are especially useful for function calls that are not *expensive* [CS99]. A discussion of the cost of user defined functions is beyond the scope of this dissertation.

There is one more technical detail to deal with when going beyond NEXT: the topmost expression in a query or a view may not always be a FLWR, hence we cannot compile it into a tree of **groupby** -blocks. For the queries, this is easy to deal with: provided there are no variables bound in the prolog, we can just rewrite each of the top-most **groupby** -blocks and then assemble them back in the enclosing expression. For the views, if the expression enclosing the blocks is invertible, we can deconstruct it (by applying the inverse) and use each upper block as a separate view. If it is not invertible, then the view can only be used as a cached function call, i.e. it can only be used when rewriting a call to the same function and such that the arguments are equivalent, via mappings, to the arguments of the cached call.

The soundness of the rewriting algorithm for NEXT+ sketched above follows from the correctness of the NextRewrite procedure, as each NEXT fragment is treated separate. One consequence of having two levels of nesting is that the top groupby block of a NEXT fragment may have free variables, whose bindings come from the parent fragment. But this only means that the initial call to ExpandBlock should not take the empty mapping as argument, but the mapping of the variables bound in the upper fragment, similar to how blocks nested in the return clause get bindings for their free variables. When mapping blocks of the NEXT fragment, the only thing that changes is that some nodes may represent function calls. Those nodes need either to be matched recursively by similar

function calls from the views, or at least their arguments need to be equivalent to NEXT fragments of the views.

2.5.2 Exploiting Function Properties

As XQuery is a Turing complete language, there is no hope for a rewriting algorithm with completeness guarantees. The existence of a rewriting is undecidability even for fragments of the language (see Corollary 2.5.1). But the applicability of our tool can be ameliorated by incorporating more information about the semantics of the functions used in the queries. Capturing semantic properties can help both to find out new views that could potentially be used in a rewriting (by inferring new matches) and to deconstruct the output of the views in cases when they return results of function calls (which is equivalent to inverting the return of the view).

Extended Matching We propose to introduce function properties through user-defined equivalences between XQuery expressions. The intended meaning of such a property is that any subexpression of a query which matches *syntactically* one of the sides of the equivalence can be replaced by the other side, keeping the same bindings for the free variables. From the semantic point of view, function properties are axioms that delimit classes of functions. Thus, certain rewritings that were not correct under the assumption of uninterpreted functions may become valid when additional axioms are considered.

In general, equivalences can be interpreted as encoding transformations in both directions, left to right and right to left. To avoid rewriting into non well-formed expressions, we do not allow transformations that introduce new free variables: for instance in a left to right transformation, all the variables on the right side must have appeared on the left side.

The following example provides a flavor of how information regarding functions can be captured and used. Consider the query

for $a **in document** (*"G.xml"*)/a

for $b **in uppercase**($a/b$) **return** *expr*($b$)

where *exp* is some expression depending on b. If **uppercase** is treated as uninter-

preted, then the view

$U :=$ **document** (*"G.xml"*)$/a/b$

is dismissed as unusable since there is no match from the view to the query (U outputs b nodes, while the first loop of the query needs to extract a nodes that are passed to the expression inside the **uppercase** function call). Now assume that we extend the rewriter with the following rule stating that **uppercase** is applied independently to all members of its argument:

$$\textbf{uppercase}(E) \equiv \textbf{for } \$v \textbf{ in } E \textbf{ return uppercase}(\$v) \tag{2.1}$$

The simple application of this rule and of the rules in our normalizer transforms the query into

for $\$a$ **in document** (*"G.xml"*)$/a$

for $\$x$ **in** $\$a/b$

for $\$b$ **in uppercase**($\$x$) **return** *expr*($\$b$)

thus exposing the match with the view pattern and leading to the rewriting

for $\$a$ **in document** (*"U"*)$/b$

for $\$b$ **in uppercase**($\$a$) **return** *expr*($\$b$)

Equivalence (2.1) expresses the property that applying the **uppercase** function to a sequence is the same as concatenating the results of the function on each item in the sequence, similar to using *map* plus *flatten* in a functional programming language. Other functions such as **lowercase**, **normalize-space** etc. may exhibit similar properties.

Consider now a view defined by a NEXT+ query

$W :=$ **for** $\$y$ **in document** (*"G.xml"*)$/a/b$, $\$z$ **in uppercase**($\$y$)
return $\langle r\rangle\{\$y, \$z\}\langle/r\rangle$

In this case, the $\$z$ variable is bound to results returned by the function and then used in the return clause. If **uppercase** does not return elements or if the label of the elements returned by **uppercase** is not known, then we are not able to build the navigation into the view output using only tree patterns. As we noticed in section 2.4.1, inverting return functions may require using additional XQuery constructs, such as positional expressions. Assuming that a call to **uppercase**($\$y$) returns only one item, a possible rewriting using W is:

for $u **in document** (*"W"*)/r[2]
return *expr*($u)

that uses a positional predicate in order to select the second node from the output

In general, we can easily invert return functions in which variables are bound either to path expressions or to functions returning singletons and all other expressions are enclosed in tags, such as a view of the form:

for $x **in uppercase**(**document** (*"G.xml"*)/a)
return ⟨r⟩{$x}⟨q⟩{*expr*($x)}⟨q/⟩⟨/r⟩

We may need positional expressions to invert such views in two cases. One is when the same label appears twice in the same return clause, either in tags of element constructors (as in the example from section 2.4.1) or because there are variables bound to the same tag. The second case is when variables are bound to results of function calls, as in the view W above.

Other examples of functional equivalences that appear in practice involve properties of arithmetic operators. For instance, commutativity of multiplication

$$\$x * \$y \equiv \$y * \$x \tag{2.2}$$

is needed to enable using the view

for $p **in** $doc/papers/sigmod/paper, $t **in** $doc/papers/taxpercentage
return ⟨taxes⟩{$p, $t * $p/price}⟨/taxes⟩

for rewriting the query

for $p **in** $doc/papers/sigmod/paper, $t **in** $doc/papers/taxpercentage
return ⟨total⟩{$p/price * $t}⟨/total⟩

Other common properties such as associativity, symmetry, antisymmetry and others could be easily expressed by such equivalences.

It is thus natural for the user to specify a set of equivalences that may interact in a complex way and that could even trigger each other recursively. This leads to a common problem in rule based system, that of deciding when to stop a potentially infinite sequence of rewritings or of imposing syntactic restrictions that would guarantee termination. Choosing between these approaches would require studying typical use cases, and finding the best trade-off between increasing expressivity and maintaining rewriting time within acceptable bounds.

Inverting the Return Functions In NEXT, we only allow return functions that can be inverted by path navigation. By adding functional equivalences, we can cover a wider range of *inverse functions*, similar to the ones implemented in the BEA AquaLogic DSP [BCL+06, BEA]. As an example, consider a function mkname with arity 2, building a full-name out of the last-name and first-name of a person. Suppose there is also a function lname which can retrieve the last-name from the full-name:

$$\mathsf{lname}(\mathsf{mkname}(\$\mathsf{ln}, \$\mathsf{fn})) \equiv \$\mathsf{ln} \tag{2.3}$$

Then one could use a view that outputs full-names

for $\$l$ **in** $\$p$/lastname/text()
for $\$f$ **in** $\$p$/firstname/text()
return ⟨fullname⟩{mkname($\$l, \$f$)}⟨/fullname⟩

in order to rewrite the query asking for last-names

for $\$l$ **in** $\$p$[firstname]/lastname/text() **return** $\$l$

by deconstructing the full-names

for $\$n$ **in** $\$docV$/fullname/text() **return** lname($\$n$)

Theorem 2.5.2. *It is undecidable whether, for a given set of function properties, there is a mapping from a NEXT+ expression into another.*

Proof: The proof follows by reduction from the Post Correspondence Problem which is known to be undecidable [UH79] for a vocabulary Σ with at least two symbols.

For a given PCP instance $(\{u_i\}_{1\le i\le n}, \{w_i\}_{1\le i\le n})$ over Σ, we create the functions $u()$ and $w()$ satisfying the equivalences

$$\begin{cases} u() & \equiv & u()/\alpha_i^1/\ldots/\alpha_i^j & u_i = \alpha_i^1 \ldots \alpha_i^j, 1 \le i \le n \\ u() & \equiv & \alpha_i^1/\ldots/\alpha_i^j & u_i = \alpha_i^1 \ldots \alpha_i^j, 1 \le i \le n \\ w() & \equiv & w()/\beta_i^1 \ldots/\beta_i^k & w_i = \beta_i^1 \ldots \beta_i^k, 1 \le i \le n \\ w() & \equiv & \beta_i^1/\ldots/\beta_i^k & w_i = \beta_i^1 \ldots \beta_i^k, 1 \le i \le n \end{cases}$$

■

Restrictions Because the general problem is undecidable, we restricted our attention to common use cases, such as the ones in the examples above, in order

to be able to match the custom rewriting rules more efficiently. In our optimizer, equivalences are interpreted as rewriting rules applied from left to right and we require that the left part be an expression tree rooted at a function or operator call.

Classification We identified two major classes of equivalences that are useful in practice, each of them corresponding to a different optimization stage. To all these equivalences, the restrictions mentioned above apply.

1. *normalizer extensions*: we allow the user to register extensions to the normalizer, such as Equivalence (2.1), each of them accompanied by a counter. Starting from the original expression, we run the rewriting rules for a maximum number of times bounded by each rule's counter. This style of rewritings suits optimizers on a mediator that has no cost information concerning the data sources. For cases in which cost information would be available, with small changes in the implementation, we could generate equivalent NEXT+ subplans, grouped as alternative *choice* branches, and choose the one with the best cost.

2. *extended pattern matching*: Same properties that are common to many functions, such as the commutativity from Equivalence 2.2 can be built in the module that matches NEXT+ patterns. Care must be taken to guarantee termination, and not to deteriorate performance. One simple restriction would be to include only properties described by expressions that do not bind variables and have only a small number of free variables, the same on both sides of the equivalence.

3. *inverse functions*: user-defined inverses are used, when compiling the views, to generate equivalent *choice* branches for the patterns of the inverses. For instance, in the example for Equivalence (2.3) there will be a choice between two patterns. One is an inverse pattern that simply navigates down a fullname and can only be matched by a query performing a call to mkname. The second one, as in the example, navigates down a fullname child, followed by a call to lname, producing the bindings for $\$l$.

Implementation We implemented normalizer extensions with associated counters by adding a *pre-normalization* stage. This is similar to the stages of our normalizer, but runs before them, and looks for syntactic matches of user defined rewriting rules against the query plan. A user defined rule has the form $P(free) \rightarrow C(free)$, where P and C are XQuery expressions and *free* is a set of free variables. For more clarity and ease of implementation, the free variables have to be declared in a reserved namespace, http://db.ucsd.edu/freevars, for which there is a default reserved prefix, free. The rewriter tries to map all variables of P, both free and bound, into the query, and if such a mapping m exists, it replaces the matching subexpression with an instantiation of C in which *free* are replaced by *m(free)*. And in order to avoid the capture of variables, bound variables from C are replaced by fresh new ones. The user defined normalization rules can be tested in our online demo at http://db.ucsd.edu/reform

We are currently studying the usage of inverse functions in data integration scenarios, and plan to include this type of optimizations into our rewriting engine.

2.6 Taking Order into Account

We know from Corollary 2.5.1 that the rewriting problem is undecidable for XQuery with list semantics, due to the undecidability of query emptiness [Van05]. This comes to no surprise, as an additional consequence of results from [Van05] is that query containment for XQuery with ordered semantics is undecidable.

We extended our algorithm to produce rewritings that take order into account, giving out completeness.

The order of an XQuery's result corresponds to the order in which variable bindings are generated in the binding stage. Normalization preserves ordering: a clause **groupby** $\$x, \$y, \$z$ can only be obtained if, before normalization, the **for** loop binding $\$x$ appears before the $\$y$ loop, which in turn appears before the $\$z$ loop. Consequently, the order of the variable bindings is induced by the lexicographic ordering of the variables in the **groupby** lists. For instance, clause **groupby** $\$x, \$y, \$z$ orders the triples of bindings first by the document order of

the bindings of $\$x$, breaking ties by the order of $\$y$ bindings, whose ties are broken by the order of the $\$z$ bindings.

When ordered rewritings are sought, we must preserve the initial ordering of the query's **groupby** variable list $\bar{x}_Q$ when replacing them with view variables $\bar{x}_V$. To this end, we search for an ordering of the view access paths which imposes on $\bar{x}_V$ the order desired for $\bar{x}_Q$. This is not always possible, as illustrated by Example 2.6.1, which shows a case with an unordered but no ordered rewriting.

EXAMPLE 2.6.1. *View V_1 below provides an access path for variables $\$x, \$z$ of query Q, and V_2 an access path for variable $\$y$. Q has no ordered rewriting using V_1 and V_2, since the two possible orderings of the view access paths yield the* **groupby** *lists $\$x, \$z, \$y$ and $\$y, \$x, \$z$, but not the desired $\$x, \$y, \$z$.*

Q: **for** *$x* **in** *path$_1$/x,*
$y **in** *path$_2$/y,*
$z **in** *path$_3$/z*
groupby *$x,$y,$z*
return *E($x,$y,$z)*

V_1: **for** *$x* **in** *path$_1$/x, $z* **in** *path$_3$/z*
groupby *$x,$z* **return** $\langle a\rangle\{\$x,\$z\}\langle/a\rangle$

V_2: **for** *$y* **in** *path$_2$/y*
groupby *$y* **return** $\langle b\rangle\$\{y\}\langle/b\rangle$

However, if we change the order in Q's groupby clause to be ($x,$z,$y), then a rewriting respecting order is:

R: **for** *$x* **in document** *("$V_1$")/a/x*
$z **in document** *("$V_1$")/a/z*
$y **in document** *("$V_2$")/b/y*
groupby *$x,$z,$y*
return *E($x,$y,$z)*

Order preservation is considered with respect to the document order in the input of the original query. More precisely, let us consider the query Q, with groupby-list $[x_1 \dots x_n]$ which is rewritten using the views $\{V_i\}$ with (invertible) return functions $\{f_i\}$, into R with groupby-list $[y_1 \dots y_n]$. The map $\text{copied}(y_j) = (V_k, v)$ records that variable y_j was created by copying variable v from the view V_k. Then we ask that the bindings for $[x_1 \dots x_n]$ be the same as those for $[g_i(y_1) \dots g_i(y_n)]$, where, for $\text{copied}(y_j) = (V_k, v)$, $g_i(y_j) = f_k^{-1}(v)$.

Order can be enforced by modifying the phase 2 of the algorithm (2.4.2) such that it searches for a rewriting that respects the way group-by lists should be assembled. If none is found, the algorithm fails.

Theorem 2.6.1. (Soundness of ordered rewritings) *The modified algorithm that enforces an ordering of the view access paths produces an equivalent rewriting under the ordered semantics or it fails.*

Proof: If the ordering is required, groupby-expressions corresponding to the same access path are adjacent and form a partition of the groupby-list of the rewriting R, hence also a partition of the columns of the binding table.

To each component, we associate a list of fragments obtained by grouping each column by the value of the immediately previous field and then keeping only the values for the fields of the component. We can prove the conclusion of the theorem by induction on the number of components of R's table. Inside the first fragment, binding values are obtained as in the original query Q, because they are all produced by the same access path (which has a mapping into Q). The same argument applies for the k^{th} component, supposing the first $k-1$ preserved the order of the binding table of Q. ■

Let us also note that an unordered rewriting that respects the constraints upon assembling group-by lists is always a valid ordered rewriting. Phase 3 of the algorithm does not perform any test for enforcing order and any rewriting produced by phase 2 that passes the equivalence check is accepted.

2.7 Optimizations

The NEXT rewriting algorithm tries to exploit as much information as possible by looking at all the ways the views match into the query. This may lead to rewritings that contain redundancies, some of which can be eliminated by employing the minimization techniques from [DPX04]. However, that type of minimization can only treat the rewritten query in isolation. It misses redundancies across view queries used in the same rewriting.

One useful optimization is to discard navigation paths that are present both in the view and in the query and do not have any output expressions. The argument for the soundness of this transformation is that all the data in the view output conforms to the navigation conditions in the view query, hence it is redundant to check them again.

As a simple example, consider the query

let $\$doc :=$ **document** ("$DBLP.xml$")
for $\$a$ **in distinct-values**($\$doc//paper[review]/author$)
return $\$a$

over the view V from section 2.5.1

The obtained rewriting would be

for $\$a2$ **in distinct-values**($\$docV/feedback[review]/authors/author$)
return $\$a$

However, by looking at query V, we notice that the *review* predicate is always true, because, by construction, any *feedback* element contains a *review*. Therefore, one can produce the potentially more efficient query

for $\$a2$ **in distinct-values**($\$docV/feedback/authors/author$)
return $\$a$

Another, more ambitious optimization is to detect redundant navigational predicates that appear because of redundant paths across views. Suppose that, in addition to the view V, there is also a view V' available:

```
let $doc := document('DBLP.xml')
for  $r in $doc//paper/review                                   (V')
return  ⟨feedback⟩{ $r }⟨/feedback⟩
```

The rewriting of the query from section 2.5.1 using V and V' is:

for $\$f$ **in document** $(V)/feedback, \$r$ **in** $\$f/review$
for $\$f'$ **in document** $(V')/feedback, \$r'$ **in** $\$f'/review$
where $count(\$f/authors/author) > 1$ **groupby** $[\$r]$ **return** $\$r$

But the second for-clause, ranging over the output of V' can be discarded, as the *reviews* in **document** (V') are exactly the same as the ones in **document** (V).

2.8 Experimental Evaluation

As shown in Section 2.4.5, the complexity of our algorithm is determined by the pattern width, a measure which is typically much smaller than the query size. Our experiments show that other factors such as query size and number of views do not affect the rewriting performance significantly, allowing algorithm NextRewrite to scale up to large numbers of views and large queries.

We implemented a generator of synthetic queries and views which enables us to control the following parameters: the nesting depth d of the query, the breadth b (see below) of the patterns in each **groupby** block, and the number of views.

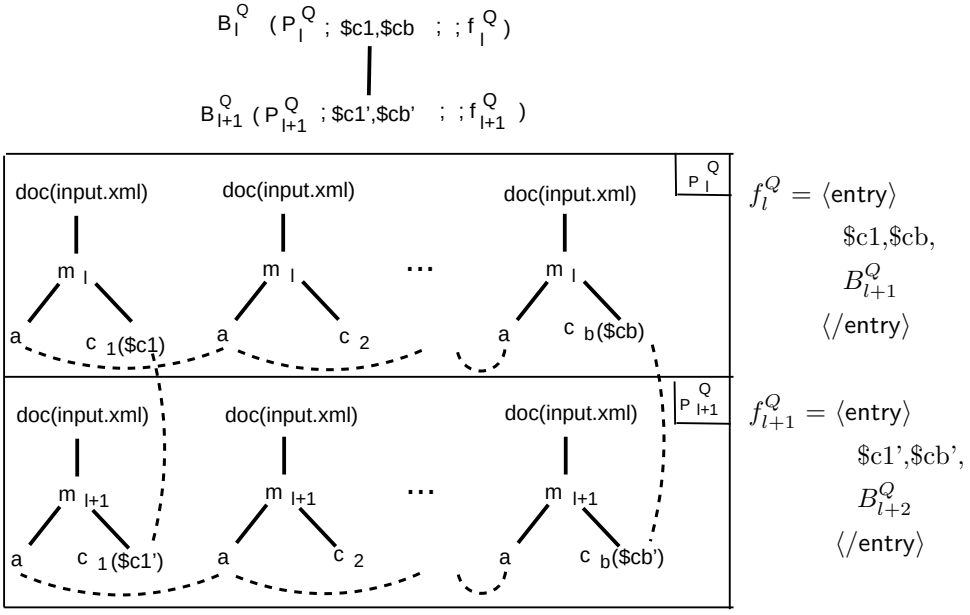


Figure 2.19: General form of synthetic queries and views

The queries. For a given value of d and b, the generator outputs a query $Q_{d,b}$ as follows. $Q_{d,b}$ has d **groupby** blocks $B_1^Q \ldots B_d^Q$, such that B_{l+1}^Q is nested within B_l^Q for each $l \in [1 \ldots d-1]$. The patterns P_l^Q and P_{l+1}^Q of blocks B_l^Q,

respectively B^Q_{l+1} are shown in Figure 2.19, using the NEXT notation. They consist of b basic patterns, where the jth basic pattern navigates from the document root to an m_l child, from there to an a and a c_j child. In the figure, we only show the variable $\$c_j$ bound to this latter child. We call the *breadth of a block* the number of basic patterns it contains. The basic patterns are chained via value-equalities between the variables binding to a elements. Variables $\$c_1$ and $\$c_b$ are the groupby-value variables of block B^Q_l, and are also output by the return function f^Q_l. These variables also appear as free variables in block B^Q_{l+1}, which performs a value-equality join between them and its own groupby-value variables, $\$c'_1$ and $\$c'_b$.

The views. We note that the detection of views which are irrelevant to the query can be done very fast, by establishing the absence of mappings from view into query. This can be detected early, as soon as some internal operator of the view pattern plan returns an empty answer. Hence an evaluation of how the algorithm scales with the number of irrelevant views would produce excellent results but would not test the real challenges. We avoid irrelevant views by generating views exclusively from subpatterns of $Q_{b,d}$.

We start with a view that is identical to the query, and we recursively split it into smaller views by alternatively halving its depth (the depth of the **groupby** tree) and breadth (the number of basic patterns in each **groupby** block). At each recursive step, the pattern of the views at each level corresponds to a subpattern of the original query. Towards a realistic scenario, we force the patterns of the views to overlap: when obtaining two new blocks B_1 and B_2 by halving a view block on its breadth (B_1 is the left half), we extend B_1's pattern with a copy of the leftmost basic pattern of B_2.

By construction of the views, the query always has a rewriting.

The platform. Our experiments were all run on a Pentium 4 2.80GHz running Windows XP with 1GB RAM. The algorithm was implemented in Java.

The measurements. Our experiments show very encouraging results. Figure 2.20 shows the results of running experiments on patterns as described above for a number of views between 1 and 128. We display the results for four

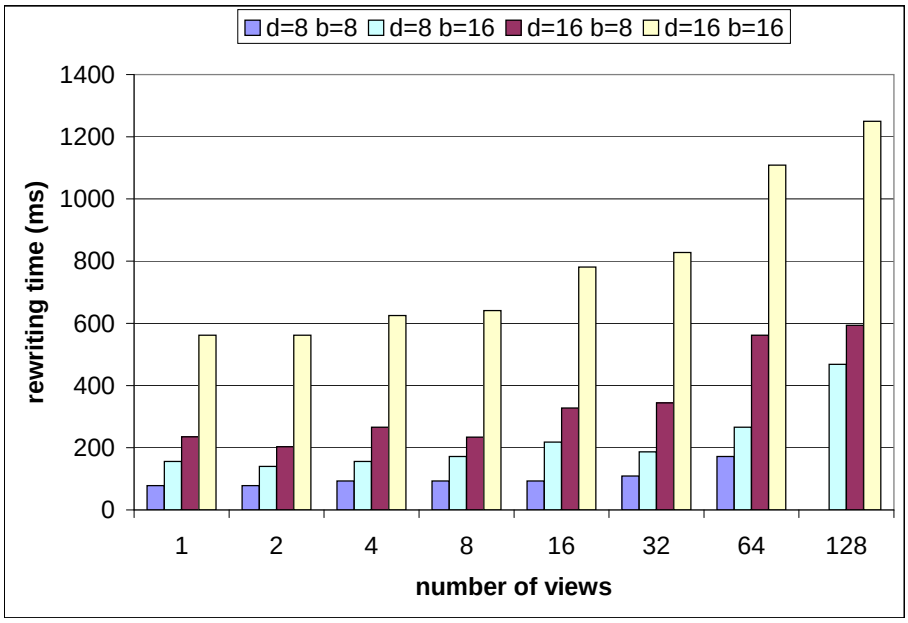


Figure 2.20: Rewriting times for increasing number of views.

configurations, corresponding to the queries $Q_{8,8}, Q_{8,16}, Q_{16,8}, Q_{16,16}$. In particular, $Q_{16,16}$ contains 16 nested blocks, each of whose patterns contains 16 value-equality conditions and binds 48 variables of which 2 are groupby-value variables. In each configuration, we measure the rewriting time when increasing the number of views by successive splits as described above. For 128 views, the test runs in 594 ms for a query with 16 nested levels and a breadth of 8 basic patterns per level. For the query of depth 16 and breadth 16 and the same number of views, we measure a rewriting time of 1250 ms.

Conclusions. While we intend to conduct experiments on real-life queries and views (the challenge there is to collect query and view specimens actually deployed in applications), our preliminary experiments are quite encouraging. They show that the algorithm performs well for large queries and large numbers of overlapping views. The reasons for this performance is the exploitation of the underlying tree structure for quickly finding mappings. By not considering irrelevant views, we generated the worst case scenario for our algorithm in order to "stress-test" it. In practice, we expect much better behavior, as many views will

be irrelevant and ruled out immediately.

2.9 Related Work

In XQuery stream processing, [DFK+04] identifies common XQuery subexpressions and memoizes in a cache to avoid redundant evaluation. This can be seen as rewriting the original XQuery using the views in the cache. The test for pattern equivalence is based on expression isomorphism and thus trades rewriting opportunities for engineering simplicity. As we discussed, one can maximize rewriting opportunities within the NEXT/OptXQuery set, while retaining syntactic isomorphism for NEXT+/XQuery. NEXT rewriting allows us to match query and view navigation *across* XPath sub-expressions, regardless of whether it appears in the **for** or the **where** clause of a query, or as **distinct-values** argument. None of these matches are supported by pure syntactic isomorphism.

[BÖB+04, XÖ05] rewrite only the XPath subexpressions of XQueries using materialized XPath indexes and thus do not face the problems caused by nesting, equalities, XML construction in the view output, all of which we address in this chapter.

The type of nesting we address here corresponds in the SQL relational case to the illegal nesting in the SELECT clause, and was therefore not a focus of SQL optimization. While this nesting is allowed in OQL, the development of complete algorithms for rewriting OQL queries using views is precluded by the fact that checking equivalence of nested OQL queries is an open problem even for the idealized conjunctive OQL sublanguage of [LS97]. We are aware of only one particular case of view-based OQL rewriting (which however does not involve nesting), namely rewriting OQL paths using path indexes [Val87, KM90]. Note that other kinds of XQuery or OQL nesting (within the FOR and WHERE clauses) are easier to deal with, as it is in most cases translated away using normalization rules such as in Agora [MFK01] (adopted in our work as well), and the normalization from [DPX04] for moving nested **some** loops from the **where** clause into the **for** clause (see NEXT_Q).

[PV99] rewrites semistructured queries using semistructured views in the context of the semistructured OEM data model. The copy semantics of XQuery's construction operators rule out the use of database id's for assembling the view data into the query result, as done in [PV99].

In the context of data integration, [YP04] rewrites XQueries under open-world, "certain answer" semantics using source-to-target constraints. The work does not address equivalent rewritings in a closed world, which is our setting. The two settings requires very different algorithms even in the relational case.

In XML publishing, the Agora [MFK01] and MARS [DT03a] systems both rewrite XQueries using publishing views, but indirectly via a reduction to rewriting relational queries using relational views (Agora) or constraints (MARS). Consequently, they do not address list and bag semantics. The strength of the two approaches is to allow mixing of relational and XML models by reducing the XQuery treatment to a relational one. A benefit of our NEXT-based approach is that, by exploiting the underlying tree pattern structure, we achieve faster algorithms for pattern matching. The relational reduction misses this opportunity. Moreover, Agora and MARS first decorrelate queries into unnested queries. Each unnested query is then individually rewritten and evaluated using relational techniques and the results are put together using an outer join. Hence the solution forces the processor to use decorrelation and outer joins at the physical level. In contrast, we provide a tool which inputs XQuery and outputs an XQuery rewriting, thus imposing no requirements on the underlying engine, with obvious benefits of portability and of allowing the cost-based optimizer to subsequently choose a physical level plan.

The equivalence checker is a basic building block for any rewriting algorithm. There is a significant body of work on equivalence of XPath (tree patterns) ([MS04, AYCLS02, Ram02, DT03b, Woo03, NS03, FFM03]). The only work we are aware of on equivalence of nested XML queries was reported in [DPX04] and adopted in this dissertation. While not complete for checking equivalence of entire queries, the algorithm of [DPX04] is complete for checking equivalence of *binding stages*, which is ultimately what we want to rewrite. [DHT04a] completely

solves checking containment for a class of nested XML queries subsumed by ours. However, it addresses a containment flavor based on homomorphic embedding of the resulting XML trees. Unfortunately, equivalence is not reducible to this kind of containment (two queries may be contained in each other without being equivalent). Moreover, the test for this containment flavor has inherently higher complexity (Π_2^p-complete if applied to NEXT patterns) than for the binding stage containment we use (NP-complete in the pattern width). One may wonder how we check binding stage equivalence under bag semantics, given that containment under bag semantics is open for conjunctive queries (but Π_2^p-hard) and undecidable for unions thereof [CV93]. Fortunately, in XML nodes have unique identities, and for a NEXT query, a bag of element values corresponds to a *set* of their identities, which allows us to reduce the test to set containment of tuples consisting of values and identities. Finally, we do not inherit the open problem of checking equivalence for OQL [LS97]. As shown in [LS97], equivalence and containment are not inter-reducible, except for OQL queries yielding VERSO relations [AHV95]. These are essentially obtained as the result of grouping, and therefore are produced by the bind stage of NEXT queries.

Previous work on containment and minimization of tree patterns has shown that, if wildcard child navigation is ruled out, then containment is characterized by a tree pattern mapping from the containing to the contained pattern [MS04, AYCLS02, Ram02]. Subsequent work on minimization of NEXT queries [DPX04], extended this result to NEXT patterns. We use the NEXT containment mappings in our rewriting work for deciding query equivalence.

APPENDIX TO THE CHAPTER

2.A Normalization Rules Beyond OptXQuery

for $\mathbb{B}_1, \$V_1$ **in** (**for** $\mathbb{B}_2$(**where** C_1)? **return** E_1), $\mathbb{B}_3$
(**where** C_2)?
(R15) **return** E_2
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}_2, \$V_1$ **in** $E_1, \mathbb{B}_3$ **where** C_1 **and** C_2 **return** E_2

for $\mathbb{B}_1, \$V_1$ **in** (**for** $\mathbb{B}_2$(**where** C_1)? **return** E_1), $\mathbb{B}_3$
(**where** C_2)? **return** E_2
$\mapsto$
(R16)
for $\mathbb{B}_1$**return**
  for $\mathbb{B}_2, \$V_1$ **in** E_1(**where** C_1)? **return**
    for $\mathbb{B}_3$(**where** C_2)? **return** E_2

for $\$V$ **in** $\langle e \rangle \{E_1\} \langle /e \rangle$**where** C **return** E_2
$\mapsto$
(R17) **if** C' **then** (E_2') **else** ()
$C' = \theta_{\$V \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}(C)$
$E_2' = \theta_{\$V \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}(E_2)$

for $\mathbb{B}_1$, **let** $\$V := E_1, \mathbb{B}_2$
(**where** C)? **return** E_2
$\mapsto$
for $\mathbb{B}_1, \mathbb{B}'_2$
(R18) (**where** C')? **return** (E_2')
(* $\theta_{\$V \mapsto E_1}(E_2)$ substitutes E_1 for $\$V$ in E_2 *)
$C' = \theta_{\$V \mapsto E_1}(C)$
$O' = \theta_{\$V \mapsto E_1}(O)$
$\mathbb{B}'_2 = \theta_{\$V \mapsto E_1}(\mathbb{B}_2)$
$E_2' = \theta_{\$V \mapsto E_1}(E_2)$

for $\$V_1$ **in** $\langle e \rangle \{E_1\} \langle /e \rangle$ **return** E_2
(R19) $\mapsto$
$\theta_{\$V_1 \mapsto \langle e \rangle \{E_1\} \langle /e \rangle}(E_2)$

for $\mathbb{B}_1, \$X$ **in** $E_1/b[P_1][P_2]\ldots[P_n], \mathbb{B}_2$
(**where** C)?
return E_2
$\mapsto$

(R20)

for $\mathbb{B}_1, \$X$ **in** $E_1/b[P_2]\ldots[P_n], \mathbb{B}_2$
where $bool(P_1')$ **and** C **return** E_2
(* If P_1 returns a boolean value. P_1' is the result of replacing every *contextual path expression p* in P_1 with $\$X/p$. A contextual path expression p in P_1 is a path expression anywhere in P_1 that starts with a QName and is not inside any predicate. $bool(P) = not\_empty(P)$ if P is a single XPath expression; otherwise $bool(P) = P$ *)

for $\mathbb{B}_1, \$X$ **in** $E_1/b[P_1][P_2]\ldots[P_n], \mathbb{B}_2$
(**where** C)? **return** E_2
$\mapsto$

(R21)

for $\mathbb{B}_1, \$X$ **in** $E_1/b[P_2]\ldots[P_n], \mathbb{B}_2$
where position $(\$X) = P_1'$ **and** C**return** E_2
(* If P_1 returns a numeric value. **position** ($X) returns the position in the sequence where the value of $X is bound. P_1' is the result of replacing every *contextual path expression p* in P_1 with $\$X/p$. A contextual path expression p in P_1 is a path expression anywhere in P_1 that starts with a QName and is not inside any predicate *)

$PathExpr$ **op** $Expr$
$\mapsto$

(R22)

some $\$V$ **in** $PathExpr$ **satisfies** $\$V$ **op** $Expr$
op is one of =, ! =, <, <=, >, and >= that are existentially quantified comparisons.

for $\$V$ **in** $(\$V' \mid \langle c\rangle\{E\}\langle/c\rangle)$**where** C **groupby** G
return E

(G13)

$\mapsto$
$\theta_{\$V \mapsto (\$V' \mid \langle c\rangle\{E\}\langle/c\rangle)}$(**if** C **then** (E') **else** ())
(* if $V' is not defined by **let** . $E' = \theta_{\$X_i \mapsto exp_i}(E)$ for every "$expr_i$ *as* $\$X_i$" in G. *)

Chapter 2 is currently being prepared for submission for publication of the material. Onose, Nicola; Deutsch, Alin; Papakonstantinou, Yannis; Curtmola,

Emiran; Xu, Yu. The dissertation author was the primary investigator and author of this material.

Chapter 2, in part, is a revised reprint of the material published in SIGMOD 2006. Onose, Nicola; Deutsch, Alin; Papakonstantinou, Yannis; Curtmola, Emiran. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Rewriting XPath Using an Intersection of XPath Views

ABSTRACT OF THE CHAPTER

In this chapter we consider views and user queries that perform navigation (XPath) and study the impact of introducing the intersection operator in the language of the query $\mathcal{L}_Q$, of the views $\mathcal{L}_\mathcal{V}$ and of the rewriting $\mathcal{L}_R$.

The standard approach for rewriting XPath queries using views consists in navigating inside a view's output, thus allowing the usage of only one view in the rewritten query. Algorithms for richer classes of XPath rewritings, using intersection or joins on node identifiers, have been proposed, but they either lack completeness guarantees, or require additional information about the data. I identify the tightest restrictions under which an XPath can be rewritten in polynomial time using an intersection of views and propose an algorithm that works for any documents or type of identifiers. As an additional contribution, I analyze the complexity of the related problem of deciding if an XPath with intersection can be equivalently rewritten as one without intersection or union.

3.1 Introduction

In Chapter 2, all XQuery constructs that are not NEXT primitives are treated as uninterpreted functions. However, intersecting views to obtain a rewriting has been heavily used in previous work on relational queries. Initially, intersecting XML views was not possible, due to semantics issues, but it was enabled by recent developments. Hence it is a natural extension to study intersection as a primitive operation.

In this chapter we study the problem of rewriting queries using views (i.e., support for listed sets of services) for queries and views that perform navigation (corresponding to the XPath fragment of XQuery) and rewritings that can additionally perform intersection. Then we go further and allow intersection in the queries and views also. The problem of rewriting queries using an intersection of views for the entire XQuery is left as future work.

XPath [CD99] is the standard for navigational queries over XML data and is widely used, either directly, or as part of more complex languages (such as XQuery [BCF$^+$]). Early research [XÖ05, MS05, TZ05, YLH03] studied the problem of equivalently rewriting an XPath by navigating inside a *single* materialized XPath view. This is the only kind of rewritings supported when the query cache can only store or can only obtain *copies* of the XML elements in the query answer, and so the original node identities are lost.

We have recently witnessed an industrial trend towards enhancing XPath queries with the ability to expose node identifiers and exploit them using intersection of node sets (via identity-based equality). This trend is supported by such systems as [BÖB$^+$04] and has culminated in the new XPath 2.0 standard [BBC$^+$07], which adds intersection as a first-class primitive. In a more general setting, intersection between collections of nodes can be based not only on physical node identifiers, but also on logical ids or keys. Research on keys for XML, such as the ones proposed in [BDF$^+$01], led to the introduction of a special *key* construct in the XML Schema [FW04] standard, which allows to uniquely identify a node based on the result of an XPath expression.

This development enables for the first time multiple-view rewritings ob-

tained by intersecting several materialized view results. The single-view rewritings considered in early XPath research have only limited benefit, as many queries with no single-view rewriting can be rewritten using multiple views.

Our work is the first to characterize the complexity of the intersection-aware rewriting problem. We identify tight restrictions under which sound and complete rewriting can be performed efficiently, i.e. in polynomial time, and beyond which the problem becomes intractable (coNP hard). These restrictions are practically interesting as they permit expressive queries and views with descendant navigation and path filter predicates.

As a side-effect of our study of rewriting, we analyze the complexity of the problem of deciding if an XPath with intersection can be equivalently rewritten as one without intersection or union, case in which we say it is *union-free*. We also study the effect of intersection on the complexity of containment of XPath 2.0 queries.

Prior work on XPath containment derived coNP lower bounds in the presence of wildcard child navigation, yet showed PTIME for tree patterns without wildcard [MS04]. In contrast, we show that extending wildcard-free tree patterns with intersection already leads to intractability.

Running Example. Throughout this chapter we will consider an example based on XPath queries over a digital library, which consists in a large number of publications, including scientific papers. A paper is organized into a hierarchy of sections, which may include, among other things, figures and images, usually related to the theorems and other results stated in the papers.

Let us assume that there has already been a query v_1, that retrieved all images appearing in sections with theorem statements:

$$v_1 : \mathsf{doc(\text{“}L\text{”})//paper//section[theorem]//image}$$

The result of v_1 is stored in the cache as a materialized view, rooted at an element named v_1. Later, the query processor had to answer another XPath v_2 looking for images inside (floating) figures that can be referenced:

$$v_2 : \mathsf{doc(\text{“}L\text{”})/lib/paper//section//figure[caption//label]/image}$$

The result of v_2 is not contained in that of v_1, so it was also executed and its answer cached.

Let us first look at an incoming query q_1, asking for all postscript images that appear in sections with theorems:

$$q_1 : \mathsf{doc(\text{“}L\text{”})//paper//section[theorem]//image[ps]}$$

q_1 can be easily answered by navigating inside the view v_1, using the following XPath query:

$$r_1 : \quad \mathsf{doc(\text{“}}v_1\mathsf{\text{”})}/v_1/\mathsf{image[ps]}$$

Now, consider a query q_2 looking for the files corresponding to images inside labeled figures from sections stating theorems:

$$q_2 : \mathsf{doc(\text{“}L\text{”})/lib/paper//section[theorem]//figure[caption//label]/image/file}$$

It is easy to see that q_2 cannot be answered in isolation using only v_1 or only v_2, because, for instance, there is no way to enforce that an image is both in a section having theorems and inside a labeled figure. However, by intersecting the results of the two views (assuming they both preserve the identities of the original image elements), one can build a rewriting equivalent to q_2:

$$r_2 : (\mathsf{doc(\text{“}}v_1\mathsf{\text{”})}/v_1/\mathsf{image} \cap \mathsf{doc(\text{“}}v_2\mathsf{\text{”})}/v_2/\mathsf{image})/\mathsf{file}$$

Chapter outline. The rest of the chapter is organized as follows. We discuss related work in Section 3.5. Section 3.2 introduces general notions and the rewriting problem. Section 3.3 presents our rewriting algorithm for tree pattern queries and views, when the rewriting allows an intersection of compensated views, followed by compensation. Section 3.4 extends the rewriting algorithm to queries, views and rewritings that can freely mix navigation and intersection. The proofs of completeness are in Appendix 3.A. Proofs of hardness for cases going beyond our PTIME fragments are presented in Appendix 3.B.

3.2 Preliminaries

We consider an XML document as an unranked, unordered rooted tree t modeled by a set of edges $\text{EDGES}(t)$, a set of nodes $\text{NODES}(t)$, a distinguished root node $\text{ROOT}(t)$ and a labeling function λ_t, assigning to each node a label from an infinite alphabet Σ. Every node n of a tree has a text value $\mathit{text}(n)$, possibly empty.

We consider XPath queries with child / and descendant // navigation, without wildcards. We call the resulting language XP , and define its grammar as:

$$\begin{aligned}
\mathit{apath} &::= \mathit{doc}(\text{“}\mathit{name}\text{”})/\mathit{rpath} \mid \mathit{doc}(\text{“}\mathit{name}\text{”})//\mathit{rpath} \\
\mathit{rpath} &::= \mathit{step} \mid \mathit{rpath}/\mathit{rpath} \mid \mathit{rpath}//\mathit{rpath} \\
\mathit{step} &::= \mathit{label}\ \mathit{pred} \\
\mathit{pred} &::= \epsilon \mid [\mathit{rpath}] \mid [\mathit{rpath} = C] \mid [.//\mathit{rpath}] \\
&\quad\ [.//\mathit{rpath} = C] \mid \mathit{pred}\ \mathit{pred}
\end{aligned}$$

The sub-expressions inside brackets are called *predicates*. C terminals stand for text constants.

Definition 3.2.1 (XP Semantics)**.** *The result of evaluating XP expression q over an XML tree t is defined as a binary relation over* $\text{NODES}(t)$*:*

1. $[\![\mathit{label}]\!]_t = \{(n, n') | (n, n') \in \text{EDGES}(t), \lambda_T(n') = \mathit{label}\}$
2. $[\![\mathit{pred}]\!]_t = \{n | n \in \text{NODES}(t), \mathit{pred}(n) = \mathit{true}\}$

   Let pred be defined as $[rq]$ or $[.//rq]$ and let t_n denote the subtree rooted at n in t. We say that $\mathit{pred}(n) = \mathit{true}$ iff $[\![\lambda_t(n)/rq]\!]_{t_n} \neq \oslash$ ($[\![\lambda_t(n)//rq]\!]_{t_n} \neq \oslash$, resp.). If pred is of the form $[rq = C]$ (or $[.//rq = C]$) then $\mathit{pred}(n) = \mathit{true}$ iff $\mathit{text}\left([\![\lambda_t(n)/rq]\!]_{t_n}\right) = C$ (or $\mathit{text}\left([\![\lambda_t(n)//rq]\!]_{t_n}\right) = C$, resp.).
3. $[\![\mathit{label}\ \mathit{pred}]\!]_t = \{(n, n') | (n, n') \in [\![\mathit{label}]\!]_t, n' \in [\![\mathit{pred}]\!]_t\}$
4. $[\![\mathit{rpath}_1/\mathit{rpath}_2]\!]_t = \{(n, n') | (n, n') \in [\![\mathit{rpath}_1]\!]_t \circ [\![\mathit{rpath}_2]\!]_t\}$
5. $[\![\mathit{rpath}_1//\mathit{rpath}_2]\!]_t = \{(n, n') | (n, n') \in [\![\mathit{rpath}_1]\!]_t \circ \text{EDGES}^*(t) \circ [\![\mathit{rpath}_2]\!]_t\}$

6. $[\![doc(\text{“}name\text{”})/rpath]\!]_t = \{(\text{ROOT}(t), n) \,|\, (\text{ROOT}(t), n) \in [\![rpath]\!]_t\}$

7. $[\![doc(\text{“}name\text{”})//rpath]\!]_t = \{(\text{ROOT}(t), n') \,|\, (\text{ROOT}(t), n') \in \text{EDGES}^*(t) \circ [\![rpath]\!]_t\}$.

(We suppose that doc(“name”) gives access to the document storing t.)

In the following, we will prefer an alternative representation widely used in literature, the unary *tree patterns* [MS04]:

Definition 3.2.2. *A* tree pattern *p is a non empty rooted tree, with a set of nodes* $\text{NODES}(p)$ *labeled with symbols from Σ, a distinguished node called the* output node $\text{OUT}(p)$*, and two types of edges:* child edges*, labeled by / and* descendant edges*, labeled by //. The root of p is denoted* $\text{ROOT}(p)$.
Every node n in p has a test of equality test(n) that is either the empty word ϵ, or a constant C. If n is on a path between $\text{ROOT}(p)$ *and* $\text{OUT}(p)$*, then test(n) is ϵ.*

Any *XP* expression can be translated into a tree pattern query and vice versa (see, for instance [MS04]). For a given *XP* expression q, by *pattern*(q) we denote the associated tree pattern p and by *xpath*$(p) \equiv q$ the reverse transformation.

The semantics of a tree pattern can be given using embeddings:

Definition 3.2.3. *An* embedding *of a tree pattern p into a tree t over Σ is a function e from* $\text{NODES}(p)$ *to* $\text{NODES}(t)$ *that has the following properties: (1) $e(\text{ROOT}(p)) = \text{ROOT}(t)$; (2) for any $n \in \text{NODES}(p)$, $\text{LABEL}(e(n)) = \text{LABEL}(n)$; (3) for any $n \in \text{NODES}(p)$, if test$(n) = C$ then text$(e(n)) = C$; (4) for any /-edge (n_1, n_2) in p, $(e(n_1), e(n_2))$ is an edge in t; (5) for any //-edge (n_1, n_2) in p, there is a path from $e(n_1)$ to $e(n_2)$ in t.*

The *result* of applying a tree pattern p to an XML tree t is the set:

$$\{(\text{ROOT}(t), e(\text{OUT}(p))) \,|\, e \text{ is an embedding of } p \text{ into } t\}$$

Let $XP^{\cap}$ be the extension of *XP* with respect to intersection, having a straightforward semantics. The grammar of $XP^{\cap}$ is obtained from that of *XP* by adding the rules:

$$\begin{aligned} ipath &::= cpath \mid (cpath) \mid (cpath)/rpath \mid (cpath)//rpath \\ cpath &::= apath \mid apath \cap cpath \end{aligned}$$

Definition 3.2.4 ($XP^{\cap}$ Semantics)**.** *$XP^{\cap}$ has the semantics:*

- $cpath \cap apath = [\![cpath]\!]_t \cap [\![apath]\!]_t$
- $[\![cpath/rpath]\!]_t = \{(n,n') | (n,n') \in [\![cpath]\!]_t \circ [\![rpath]\!]_t\}$
- $[\![cpath//rpath]\!]_t = \{(n,n') | (n,n') \in [\![cpath]\!]_t \circ \text{EDGES}^*(t) \circ [\![rpath]\!]_t\}$

By $XP^{\cap}$ expressions over a set of documents D we denote those that use only *apath* expressions that navigate inside the documents D. For a fragment $\mathcal{L} \subseteq XP$, by $\mathcal{L}^{\cap} \subseteq XP^{\cap}$ we denote the $XP^{\cap}$ expressions that use only *apath* expressions from $\mathcal{L}$.

Similar to the XP - tree pattern duality, we can represent $XP^{\cap}$ expressions using the more general *DAG patterns*:

Definition 3.2.5. *A* DAG pattern *d is a directed acyclic graph, with a set of nodes* $\text{NODES}(d)$ *labeled with symbols from Σ, a distinguished node called the* output node $\text{OUT}(d)$*, and two types of edges:* child edges*, labeled by / and* descendant edges*, labeled by //. d has to satisfy the property that any $n \in \text{NODES}(d)$ is accessible via a path starting from a special node* $\text{ROOT}(d)$*. In addition, all the nodes that are not on a path from* $\text{ROOT}(d)$ *to* $\text{OUT}(d)$ *(denoted* predicate nodes*) have only one incoming edge.*
Every node n in d has a test of equality test(n) that is either the empty word ϵ, or a constant C. If n is on a path between $\text{ROOT}(d)$ *and* $\text{OUT}(d)$*, then test(n) is always ϵ.*

Figure 3.1(a) gives an example of a DAG pattern. $\text{ROOT}(d)$ is the *doc*(L) node and $\text{OUT}(d)$ is the *image* node indicated by a square.

Representing $XP^{\cap}$ by DAG patterns. For a query q in $XP^{\cap}$, we construct the associated pattern $\texttt{dag}(q)$ as follows:

1. for every *apath* (XP path with no $\cap$), $\texttt{dag}(apath)$ is the tree pattern corresponding to the *apath* .
2. $\texttt{dag}(p_1 \cap p_2)$ is obtained from $\texttt{dag}(p_1)$ and $\texttt{dag}(p_2)$ as follows: (i) provided there are no labeling conflicts and both p_1 and p_2 are not empty, by coalescing $\text{ROOT}(\texttt{dag}(p_1))$ with $\text{ROOT}(\texttt{dag}(p_2))$ and $\text{OUT}(\texttt{dag}(p_1))$ with $\text{OUT}(\texttt{dag}(p_2))$ respectively, (ii) otherwise, as the empty pattern.

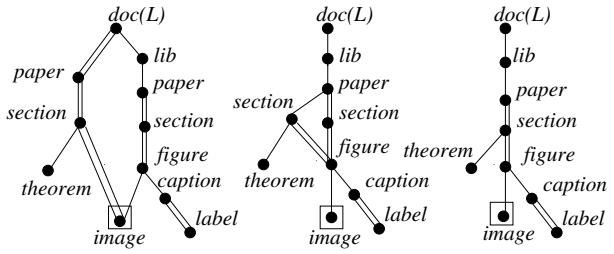


Figure 3.1: Running the rules on the example of Section 3.1

3. $\texttt{dag}(x/rpath\ )$ and $\texttt{dag}(x//rpath\ )$ are obtained as follows: (i) for non-empty x, by appending the pattern corresponding to *rpath* to $\mathrm{OUT}(\texttt{dag}(x))$ with a /- and a //-edge respectively, (ii) as x, if x is the empty pattern.

We state the following theorem, for which the proof is straightforward.

Theorem 3.2.1. *For any $q \in XP^{\cap}$ and any tree t, $q(t) = \texttt{dag}(q)(t)$.*

By a pattern from language $\mathcal{L}$ we denote any pattern built as $dag(q)$, for $q \in \mathcal{L}$. Note that a tree pattern is a DAG pattern as well. The notion of *embedding* and the semantics of a pattern can be extended in straightforward manner from trees to DAGs. In the following, unless stated otherwise, all patterns are DAG patterns.

We say that two tree patterns p_1 and p_2 are *incomparable* iff there is no containment mapping between them.

By the main branch nodes of a pattern d, $\mathrm{MBN}(d)$, we denote the set of nodes found on paths starting with $\mathrm{ROOT}(d)$ and ending with $\mathrm{OUT}(d)$. We refer to main branch paths between $\mathrm{ROOT}(d)$ and $\mathrm{OUT}(d)$ as *main branches* of d. The (unique) main branch of a tree pattern p is denoted $\mathrm{MB}(p)$. A */-pattern* is a tree pattern that has only /-edges in the main branch. We call *predicate subtree* of a pattern p any subtree of p rooted at a non-main branch node.

A *prefix* p of a tree pattern q is any tree pattern with $\mathrm{ROOT}(p) = \mathrm{ROOT}(q)$, $m = \mathrm{MB}(p)$ a subpath of $\mathrm{MB}(q)$ and having all the predicates attached to the nodes of m in q. For instance, the pattern shown in Figure 3.1(c) is a prefix of the pattern of q_2, since it has all the nodes of q_2, except for the output one.

A *lossless prefix* p of a tree pattern q is any tree pattern obtained from q by setting the output node to some other main branch node (i.e., an ancestor of $\mathrm{OUT}(q)$). Note that this means that the rest of the main branch becomes a side branch, hence a predicate.

Definition 3.2.6. *A pattern d_1 is* contained *in another pattern d_2 iff for any input tree t, $d_1(t) \subseteq d_2(t)$. We write this shortly as $d_1 \sqsubseteq d_2$. We say that d_1 is* equivalent *to d_2, and write $d_1 \equiv d_2$, iff $d_1(t) = d_2(t)$ for any input tree t.*

We say that a pattern p is *minimal* [AYCLS02] if there is no other pattern $p' \equiv p$ having less nodes than p.

Definition 3.2.7. *A* mapping *between two patterns d_1 and d_2 is a function $h : \mathrm{NODES}(d_1) \to \mathrm{NODES}(d_2)$ that satisfies the properties 2,5 of an embedding (allowing the target to be a pattern) plus three others: (6) for any $n \in \mathrm{MBN}(d_1)$, $h(n) \in \mathrm{MBN}(d_2)$; (7) for any /-edge (n_1, n_2) in d_1, $(e(n_1), e(n_2))$ is a /-edge in d_2. (8) for any $n \in \mathrm{NODES}(d_1)$, if $test(n) = C$ then $test(h(n)) = C$;*

A root-mapping *is a mapping that satisfies (1). An* output-mapping *is a mapping h such that $h(\mathrm{OUT}(p_1)) = \mathrm{OUT}(p_2)$. A* containment mapping *denotes a mapping that is simultaneously a root-mapping and an output-mapping.*

Lemma 3.2.1. *If there is a containment mapping from d_1 into d_2 then $d_2 \sqsubseteq d_1$.*

We will also consider the extension $XP^{\cup}$ of XP with respect to union, having a straightforward semantics. The grammar of $XP^{\cup}$ is obtained from that of XP by adding the rule:

$$upath ::= apath \cup apath.$$

The semantics of $XP^{\cup}$ is: $apath \cup apath = [\![apath]\!]_t \cup [\![apath]\!]_t$.

Definition 3.2.8. *Given an $XP^{\cup}$ query q corresponding to a union of tree patterns $u = p_1 \cup \cdots \cup p_n$, by the* normal form of q *(in short, $nf(q)$) we denote the equivalent query $u' = p_{i_1} \cup \cdots \cup p_{i_k}$, obtained from u by keeping only the incomparable patterns.*

We next prove that one can always reformulate a DAG pattern as a (possibly empty) union of tree patterns. As in [BFK05], a *code* is a string of symbols from Σ, alternating with either / or //.

Definition 3.2.9 (Interleaving)**.** *By the* interleavings *of a pattern d we denote any tree pattern p_i produced as follows:*

1. *choose a code i and a total onto function f_i that maps* $\mathrm{MBN}(d)$ *into* Σ*-positions of i such that:*
   (a) *for any* $n \in \mathrm{MBN}(d), \mathrm{LABEL}(f_i(n)) = \mathrm{LABEL}(n)$
   (b) *for any /-edge (n_1, n_2) in d, the code i is of the form*
   $$\ldots f_i(n_1)/f_i(n_2)\ldots,$$
   (c) *for any //-edge (n_1, n_2) in d, the code i is of the form*
   $$\ldots f_i(n_1)\ldots f_i(n_2)\ldots.$$
2. *build the smallest pattern p_i such that:*
   (a) *i is a code for the main branch* $\mathrm{MB}(p_i)$*,*
   (b) *for any $n \in \mathrm{MBN}(d)$ and its image n' in p_i (via f_i), if a predicate subtree st appears below n then a copy of st appears below n', connected by same kind of edge.*

Two nodes n_1, n_2 from $\mathrm{MBN}(d)$ *are said to be* collapsed *if $f_i(n_1) = f_i(n_2)$, with f_i as above. The tree patterns p_i thus obtained are called* interleavings *of d and we denote their set by* $\mathrm{interleave}(d)$*.*

We say that a pattern d is *satisfiable* if it is non-empty and the set *interleave(d)* is non-empty. By definition, there is always a containment mapping from a satisfiable pattern into each of its interleavings. Then, by Lemma 3.2.1, a pattern will always contain its interleavings. Similar to a result from [BFK05], it also holds that:

Lemma 3.2.2. *A tree pattern p is contained into a rooted DAG pattern d iff there is a containment mapping from d into p.*

Proof Idea. Consider the model mod'_p of p in which // edges are replaced by a sequence $/z/$ (two child edges), where z is a fresh new label. If $p \subseteq d$, then in particular $d(mod'_p) \neq \emptyset$. Since z is a new label, d can only embed a // edge in a path fragment containing z.

We will now prove one of the lemmas that lays the theoretical ground for our study:

Lemma 3.2.3. *Any DAG pattern is equivalent to the union of its interleavings.*

Proof: We have to prove containment in both directions. The fact that $\bigcup p_i$, for all $p_i \in interleaved$, is contained in d follows easily from the way interleavings are defined. For each p_i, defined by a code $i = xpath(MB(p_i))$, function f_i, from the one-to-one mapping ϕ of i-positions into main branch nodes of p_i we can built a containment mapping ψ from d into p_i as follows: for every $n \in \text{MBN}(d)$, $\psi = \phi(f_i(n))$. By definition, all the predicate subtrees of each n will have an copy image in p_i at node $\psi(n)$. Hence this allows us to define ψ on the predicate subtrees as well. Moreover, f_i preserves all the child/descendant relationships between main branch nodes. Finally, since p_i is a minimal pattern satisfying the conditions of Definition 3.2.9, it follows easily that the image of $\text{ROOT}(d)$ must be the first position in i. Similarly, the image of $\text{OUT}(()\text{d})$ must be the last position of i. This allows us to conclude that ψ is a containment mapping, and thus by Lemma 3.2.1 we have that $\bigcup_i p_i \sqsubseteq d$.

We now consider the other inclusion, from d into $\bigcup p_i$. We show that for any XML tree t and any node $n \in t$ such that $(\text{ROOT}(t), n) \in e(t)$, for some embedding e of d into t (so $e(\text{OUT}(d)) = n$), we can always find an interleaving p_i and embedding e_i of p_i in t such that $(\text{ROOT}(t), n) \in e_i(t)$. This would be enough to conclude the proof of inclusion (and equivalence).

Let p denote the linear path from $\text{ROOT}(t)$ to n (endpoints included) and let c denote the code of p. Let id denote the one-to-one mapping from p to c. Note that e gives us a mapping $id \circ e$ from $\text{MBN}(d)$ to c, such that all the child/descendant relationships between main branch nodes are accordingly translated in the ordering of c. Let c' denote the code obtained from c by: Step 1) replacing by the empty string all the positions that are not the image of some node $n' \in \text{MBN}(d)$ under $e \circ id$, Step 2) replacing any sequence of consecutive /-characters of length more than 2 (i.e.,"///...") by the slash-slash sequence (i.e, "//").

We are now ready to construct the interleaving p_i and its embedding e_i, such that $(\text{ROOT}(t), n) \in e_i(t)$.

Let us book keep by a partial function f_c the correspondence between used c positions and c' positions. Let p_i be defined by the code $i = c'$, and let f_i be

defined by $f_c \circ id \circ e$ on all the nodes in $\mathrm{MBN}(d)$. It is easy to see that i and f_i give indeed an interleaving p_i, as it obeys all the conditions and p_i is minimal. Let id'' denote the one-to-one mapping from $\mathrm{MB}(p_i)$ into c'. Now, we can define its embedding e_i into t as follows: for all main branch nodes $n' \in \mathrm{MB}(p_i)$ we have $e_i(n') = id^{-1} \circ f_c^{-1} \circ id''$. It is easy to see that for any node $n'' \in MBN(d)$ such that $n'_i = id''^{-1}(f_i(n''))$, we have $e(n'') = e_i(n')$ so all the predicate subtrees in p_i can be mapped at $e_i(n')$ for all n'.

Since t and n were chosen at random, this concludes the proof of containment for $d \sqsubseteq \bigcup_i p_i$. ■

For instance, one of the seven interleavings of d in Figure 3.1(a) is the pattern in Figure 3.1(c) and another one corresponds to the XPath

$$doc(L)/\mathsf{lib/paper//paper//section[theorem]//figure[caption[.//label]]/image}$$

The following also holds:

Lemma 3.2.4. *If a tree pattern is equivalent to a union of tree patterns, then it is equivalent to a member of the union.*

Definition 3.2.10. *Given a DAG pattern p, by the* normal form of p *we denote the union $nf(\bigcup_i p_i | p_i \in interleave(p))$.*

Note that the set of interleavings p_i of a DAG pattern p can be exponentially larger than p. Indeed, it was shown that the $XP^{\cap}$ fragment is not included in XP (i.e, the union of its interleavings cannot always be reduced to one XP query by eliminating interleavings contained in others) and that a DAG pattern may only be translatable into a union of exponentially many tree patterns (see [BFK05]).

Definition 3.2.11. *We say that a DAG pattern is* union-free *iff it is equivalent to a single tree pattern.*

By Lemmas 3.2.3 and 3.2.4, a satisfiable pattern is union-free iff it has an interleaving that contains all other possible interleavings.

The rewriting problem. Given a set of views $\mathcal{V}$, defined by XP queries over a document D, by $D_{\mathcal{V}}$ we denote the set of view documents $\{doc(\text{“}V\text{”}) | V \in \mathcal{V}\}$,

in which the topmost element is labeled with the view name. Given a query $r \in XP^{\cap}$ over the view documents $D_{\mathcal{V}}$, we define *unfold*(r) as the $XP^{\cap}$ query obtained by replacing in r each *doc*("V")$/V$ with the definition of V.

We are now ready to describe the view-based rewriting problem. Given a query q and a finite set of views $\mathcal{V}$ over D in a language $\mathcal{L} \subseteq XP$, we look for an alternative plan r, called a *rewriting*, that can be used to answer q. We define rewritings as follows:

Definition 3.2.12. *For a given document D, an XP query q and XP views $\mathcal{V}$ over D, a* rewrite plan *of q using $\mathcal{V}$ is a query $r \in XP^{\cap}$ over $D_{\mathcal{V}}$. If unfold$(r) \equiv q$, then we also say r is a* rewriting.

According to the definition above and the definition of $XP^{\cap}$, a rewriting r is of the form $\mathcal{I} = (\bigcap_{i,j} u_{ij}), \mathcal{I}/rpath$ or $\mathcal{I}//rpath$, with u_{ij} of the form *doc*("V_j")$/V_j/p_i$ or *doc*("V_j")$/V_j//p_i$. We say a rewriting r is *minimal* if all p_i and all *rpath* 's are minimal.

Lemma 3.2.5. *A rewrite plan can be evaluated over a set of view documents $D_{\mathcal{V}}$ in polynomial time in the size of $D_{\mathcal{V}}$.*

Proof: We will just sketch the main idea of the proof. Consider a rewrite plan r over a set of view documents $D_{\mathcal{V}}$. The plan itself gives an evaluation strategy that is polynomial. We start from the document nodes and navigate from each of them down to the closest intersection node. All navigations can be done in PTIME, as they can be seen equivalently as tree patterns. We can prove, by induction on the structure of r, that the input of each intersection node is polynomial, hence its input is also polynomial, because the result is always a set (arity equals 1), and it has at most as many elements as the largest of its inputs. Hence the size of each intermediate result is bounded by the size of the largest view. Since the number of steps, navigation and intersections, is constant w.r.t. t (it is proportional to the size of $|r|$), the overall computation is polynomial in $D_{\mathcal{V}}$. ■

Completeness. In the following, by saying that an algorithm is *complete for rewriting $\mathcal{L} \subseteq XP$* , we mean that it solves the rewriting problem for queries and views in $\mathcal{L}$.

3.3 Rewriting Algorithm

Our approach for testing the existence of a rewriting (algorithm REWRITE) is the following: for each rewrite plan r using views that satisfies certain conditions w.r.t the query q, we test whether its unfolding is equivalent to q. We show that the number of plans to be considered depends only on the size of (the main branch) of q, and is thus linear. Testing equivalence between the tree pattern q and a DAG pattern d corresponding to the unfolding of r will be the central task in our algorithm. As the plans/DAGs to be considered will always contain q, testing equivalence will amount to testing the opposite containment, of d into q.

However, Lemmas 3.2.3 and 3.2.4 imply that equivalence holds iff d has an interleaving p_i such that $d \equiv p_i \equiv q$. From this observation, a naïve approach for the rewrite test would be to simply compute the interleavings of $unfold(r)$ (a union of interleavings), check that this union reduces by containments to one interleaving p_i (union-freedom), and that p_i is equivalent to q. We devise an algorithm for computing the interleavings and testing union-freedom that avoids the naïve approach. It is based on a set of rewrite rules R1-R8 that simulate transformation steps of d (algorithm APPLY-RULES). Each rule application will produce an equivalent pattern that is one step closer to an interleaving that contains all others, if such a one exists. This rule-based algorithm is sound and becomes a decision procedure for union-freedom under practically relevant restrictions.

APPLY-RULES is then used in the REWRITE algorithm. While the soundness of REWRITE will follow from the soundness of APPLY-RULES, we show that it is also a decision procedure.

We next detail the algorithm that rewrites q using views $\mathcal{V}$:

REWRITE$(q, \mathcal{V})$

1 *Prefs* $\leftarrow \{(p, \{(v_i, b_i)\}) \mid v_i \in \mathcal{V}, p$ a lossless prefix of $q, b_i \in \mathrm{MB}(p)$,
$\exists$ a mapping h from $u_i = pattern(v_i)$ into $p, h(\mathrm{OUT}(u_i)) = b_i\}$
2 **for** $(p, W) \in$ *Prefs*
3 **do** let $\mathcal{V}' \leftarrow \{\mathrm{compensate}(v, p, b) \mid (v, b) \in W\}$
4 let r be the $XP^{\cap}$ query $\left(\bigcap_{v_j \in \mathcal{V}'} v_j\right)$
5 let d be the DAG corresponding to *unfold*(r)
6 APPLY-RULES(d)
7 **if** $d \sqsubseteq p$
8 **then return** $\mathrm{compensate}(r, q, \mathrm{OUT}(p))$
9 **return fail**

APPLY-RULES(d)

1 **repeat**
2 **repeat** apply R1 to d
3 **until** no change
4 **repeat** apply R2-R8 to d, in arbitrary order
5 **until** no change
6 **until** no change

For a pattern d and node $n \in \mathrm{MBN}(d)$, by $\mathrm{SP}_d(n)$ we denote the subpattern rooted at n in d. The compensate function generalizes the concatenation operation from [XÖ05], by copying extra navigation from the query into the rewrite plan. For $r \in XP^{\cap}$ and a tree pattern p, compensate(r, p, n) returns the query obtained by deleting the first symbol from $x = xpath(\mathrm{SP}_p(n))$ and concatenating the rest to r. For instance, the result of compensating $r =$ a/b with $x =$ b[c][d]/e is the concatenation of a/b and [c][d]/e, i.e. a/b[c][d]/e. At line 8, if p is q itself, compensate returns just r, because all needed navigation had already been added at 3.

We also consider two modified versions of REWRITE :

ALL-REWRITES – same code as REWRITE with the modifications: (i) replace line 2 with: (2′) **for** $(p, U) \in$ *Prefs* **for** $W \subseteq U$ (ii) remove line 9 and (iii) continue to run even when the return at line 8 is reached.

EFFICIENT-RW – same code as REWRITE , except line 7, which becomes: (7′) **if** d is a tree **then if** $d \sqsubseteq p$.

We mention that at line 3 in the code, some elements of $\mathcal{V}'$ may be redundant and can be discarded. For space reasons, we do not discuss such optimizations.

We will prove that the result of APPLY-RULES is always equivalent to the original DAG. This implies that REWRITE gives a sound algorithm for the rewriting problem, and we will show it is also a decision procedure.

3.3.1 The Rewrite Rules.

We present the rules R1-R8 as pairs formed by a test condition, which checks if the rule is applicable, and a graphical description, which shows how the rule transforms the DAG. The left-hand side of the rule description will match main branch nodes and paths in the DAG. If the matching nodes and paths verify the test conditions, then the consequent transformation is applied on them. Each transformation either (i) collapses two main branch nodes n_1, n_2 into a new node $n_{1,2}$ (which inherits the predicate subtrees, incoming and outgoing main branch edges), (ii) removes some redundant main branch nodes and edges, or (iii) appends a new predicate subtree below an existing main branch node.

Notation. We use the following notation in the graphical illustration of our rewrite rules: linear paths corresponding to part of a main branch are designated in italic by the letter p, nodes are designated by the letter n, the result of collapsing two nodes n_i, n_j will be denoted $n_{i,j}$, simple lines represent /-edges, double lines represent //-edges, simple dotted lines represent /-paths, and double dotted lines represent arbitrary paths (may have both / and //). We only represent main branch nodes or paths in the graphical description of rules (predicates are omitted). An exception is rule R5, where we refer to a subtree predicate by its XP expression $[Q]$. We refer to the tree pattern containing just a main branch path p simply by p, and to the tree pattern having p as main branch by $\mathrm{TP}_d(p)$. We represent by a rhombus main branch paths that are not followed by any / (main branch) edge. Paths include their end points.

Test Conditions. In the test conditions, we say that a pattern d is *imme-*

diately unsatisfiable if by applying to saturation Rule R1 on it we reach a pattern in which either there are two /-paths of different lengths but with the same start and end node, or there is a node with two incoming /-edges λ_1/λ and λ_2/λ, such that $\lambda_1 \neq \lambda_2$. Note that the test of immediate unsatisfiability is just a sufficient condition for the unsatisfiability of the entire DAG.

For a main branch path p in d, given by a sequence of nodes $(n_1, \ldots, n_k)$, we define $\mathrm{TP}_d(p)$ as the tree pattern having p as main branch, n_1 as root and n_k as output, plus all the predicate subtrees (from d) of the nodes of p.

Definition 3.3.1. *We say that two /-patterns p_1, p_2 are* similar *if (a) their main branches have the same code, and (b) both have root mappings into any /-pattern p_{12} built from p_1, p_2 as follows:*

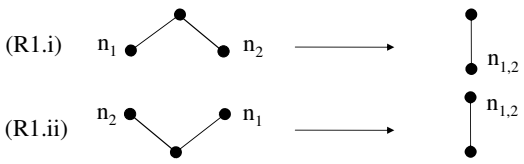
1. *choose a code i_{12} and a total onto function f_{12} that maps the nodes of $m_{12} = \mathrm{MBN}(p_1) \cup \mathrm{MBN}(p_2)$ into i_{12} such that:*
   (a) *for any node n in m_{12}, $\mathrm{LABEL}(f_{12}(n)) = \mathrm{LABEL}(n)$*
   (b) *for any /-edge (n_1, n_2) in the main branch of p_1 or p_2, the code i_{12} contains $f_{12}(n_1)/f_{12}(n_2)$*
2. *build the minimal pattern p_{12} such that:*
   (a) *i_{12} is a code for the main branch $\mathrm{MB}(p_{12})$,*
   (b) *for each node n in $\mathrm{MBN}(p_1) \cup \mathrm{MBN}(p_2)$ and its image n' in $\mathrm{MB}(p_{12})$ (via f_{12}), if a predicate subtree st appears below n then a copy of st appears below n', connected by the same kind of edge.*

Two /-patterns that are not similar are said to be *dissimilar*. We can test whether two /-patterns are similar in linear time, since there are at most $|p_1| \times |p_2|$ such p_{12} constructs.

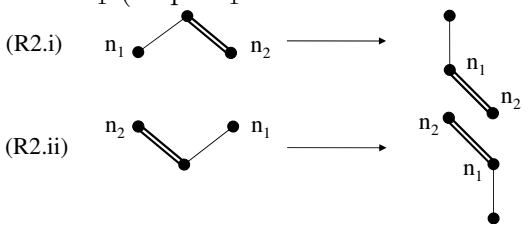
For two nodes $n_1, n_2 \in \mathrm{MBN}(d)$, such that $\lambda_d(n_1) = \lambda_d(n_2) = \lambda$, by $collapse_d(n_1, n_2)$ we denote the DAG obtained from d by replacing n_1 and n_2 with a λ-labeled node $n_{1,2}$ that inherits the incoming and outgoing edges of both n_1 and n_2. We say that two nodes n_1, n_2 are *collapsible* iff they have the same label and the DAG pattern $collapse_d(n_1, n_2)$ is not immediately unsatisfiable.

We have now all the ingredients to present the rewrite rules:

R1 This rule triggers when $\lambda_d(n_1) = \lambda_d(n_2)$



R2 This rule triggers if n_1 and n_2 are not collapsible and n_2 is not reachable from n_1 (resp. n_1 is not reachable from n_2, in the case of R2.ii).

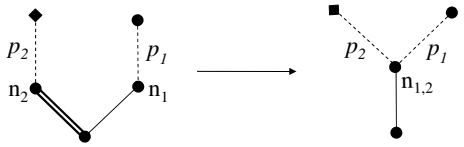


R3 i) This rule triggers if the following conditions hold:

- $p_1 \equiv p_2$,
- p_2's nodes have only one incoming main branch edge,
- $\mathrm{TP}_d(p_2)$ root-maps into $\mathrm{TP}_d(p_1)$.

R3 ii) This rule triggers if the following conditions hold:

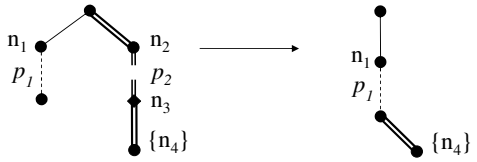
- $p_1 \equiv p_2$,
- p_2's nodes have only one outgoing main branch edge,
- $\mathrm{TP}_d(p_2)$ root-maps into $\mathrm{TP}_d(p_1)$.



R4 i) The rule triggers if the following conditions hold for all nodes n_4:

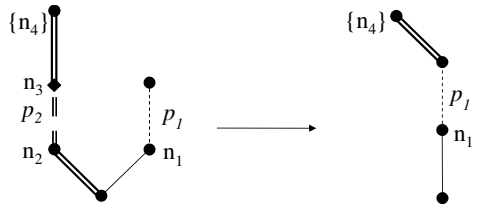
- n_3 has one incoming main branch edge, all other nodes of p_2 have one incoming and one outgoing main branch edge,
- there exists a mapping from $\mathrm{TP}_d(p_2)$ into $\mathrm{SP}_d(n_1)$, mapping all the nodes of p_2 into nodes of p_1.

- the path $p_2//n_4$ does not map into p_1.



R4 ii) This rule triggers if the following conditions hold for all nodes n_4:

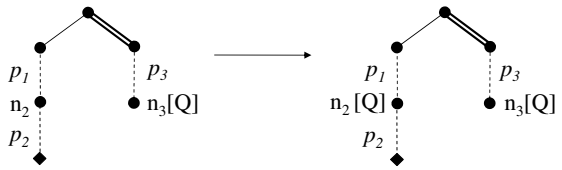
- n_3 has only one outgoing main branch edge, all the other nodes of p_2 have one incoming and one outgoing main branch edge,
- there exists a mapping from $\mathrm{TP}_d(p_2)$ into $\mathrm{TP}_d(p_1)$, mapping all the nodes of p_2 into nodes of p_1.
- the path $n_4//p_2$ does not map into p_1.



R5 This rule triggers if the following conditions hold:

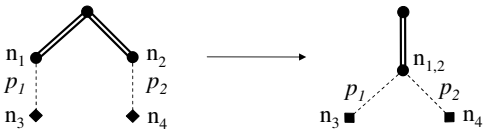
- n_2 and n_3 are collapsible and $p_1 \equiv p_3$,
- *pattern*$(\lambda_d(n_2)[Q])$ does not have a root-mapping into $\mathrm{SP}_d(n_2)$,
- for any node n_4 in p_2 such that $d' = \mathit{collapse}_d(n_4, n_3)$ is not immediately unsatisfiable, *pattern*$(\lambda_d(n_2)[Q])$ has a root mapping into $\mathrm{SP}_{d'}(n_2)$,
- if there is no path from n_3 to a node of p_2, there has to be a root-mapping from *pattern*$(\lambda_d(n_2)[Q])$ into the pattern obtained from $\mathrm{TP}_d(p_2)$ by appending $[Q]$'s pattern, via a //-edge, below the node $\mathrm{OUT}(\mathrm{TP}_d(p_2))$.

(Special case: p_1 and p_3 empty.)



R6 This rule triggers if the following conditions hold

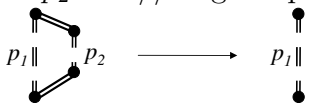
- n_3, n_4 have only one incoming main branch edge, all other nodes of p_1 and p_2 have one incoming and one outgoing main branch edge,
- $\text{TP}_d(p_1)$ and $\text{TP}_d(p_2)$ are similar.



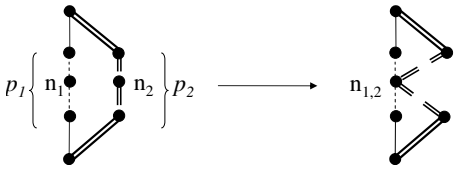
R7 This rule triggers if the following holds:

- the nodes of p_2 have only one incoming and one outgoing main branch edge,
- there exists a mapping from $\text{TP}_d(p_2)$ into $\text{TP}_d(p_1)$, such that the nodes of p_2 are mapped into nodes of p_1.

(Special case: p_2 is a //-edge in parallel with p_1.)



R8 This rule triggers if in any possible mapping of p_2 into p_1 the image of n_2 is n_1.



Some of the rules, such as R3 and R6, could safely collapse more than one node, without changing any of the results in Section 3.3.2. We opted for the current version for ease of presentation.

We illustrate in Figure 3.1 how we take the unfolding of the intersection of the views v_1 and v_2 from the example in Section 3.1 and rewrite it into a prefix of q_2 (see Figure 3.1.(c)). Then, line 8 in algorithm REWRITE adds the navigation /file and the rewriting r_2 that we intuitively discovered is computed.

3.3.2 Formal Guarantees

Using Lemma 3.3.1, we first show that algorithm REWRITE (and EFFICIENT-RW and ALL-REWRITES) is sound, i.e. it gives no false positives.

Theorem 3.3.1. *If algorithm* REWRITE *(or* EFFICIENT-RW *or* ALL-REWRITES*) returns a DAG pattern r, then unfold$(r) \equiv q$.*

To prove the theorem, we will use a few intermediary results.

Lemma 3.3.1. *The application of any of the rules from the set R1-R8 on a DAG d produces another DAG $d' \equiv d$.*

Proof: Each rule r in our set has an associated function f_r that takes a DAG d as input and outputs another DAG $f_r(d)$ that is the result of applying r to d. We divided the proof in two parts: Lemma 3.3.2 gives the containment $f_r(d) \sqsubseteq d$ and Lemma 3.3.3 $d \sqsubseteq f_r(d)$. ■

Lemma 3.3.2. *For every rule r, f_r is a containment mapping.*

Lemma 3.3.3. *For a rule r, a DAG d and a document t, if d has an embedding e in t and r is applicable to d, then $f_r(d)$ has also an embedding e' into t such that $e(\text{OUT}(d)) = e'(\text{OUT}(f_r(d)))$.*

Proof: **R1.i, R1.ii:** n_1 and n_2 belong to two different main branches, but they have a common parent n. (Remember that all paths depicted in the rules are part of main branches.) Remember also that, by the definition of a main branch, the branches of n/n_1 and n/n_2 have at least one common node below n: OUT(d). Then, in any embedding e of d into a tree t, n/n_1 and n/n_2 need to map in the same path of t and it must be true that $e(n_1) = e(n_2) = x$, where $x \in \text{NODES}(t)$. Thus there is also an embedding e' from $f_{\text{R1}}(d)$ into t that maps $n_{1,2}$ into x and is equal to e on all the other nodes.

R2.i: Let n_0 be the parent of n_1 and n_2. From the condition that n_1 and n_2 are not collapsible, we infer that either they have different labels, or the pattern obtained by trying to collapse n_1 and n_2 is immediately unsatisfiable. Both cases imply that, for an embedding e into t, we cannot have $e(n_1) = e(n_2)$. The former case is obvious. For the latter, supposing that $e(n_1) = e(n_2) = x$, we observe that the beginning of main branches under n_1 and n_2, formed only by /-edges, call them p_{n1} and p_{n2} respectively, need to map into the same nodes under x (as all main branches have at least one common ending point, OUT(d), and we are mapping

them into a tree). Then the pattern obtained by collapsing n_1 and n_2 would also have an embedding into t, that can be computed from e by equating nodes n_1 and n_2. But this contradicts the assumption that the pattern obtained by trying to collapse n_1 and n_2 is immediately unsatisfiable. Hence, for an embedding e into t, $e(n_1) \neq e(n_2)$, and, since $e(n_1)$ has to be a child of $e(n_0)$, $e(n_2)$ has to be a strict descendant of $e(n_1)$. This guarantees that $n_0/n_1//n_2$ will also map into t if d does.

R2.ii: The proof is very similar to the one for R2.i

R3.i: Let n_0 be the parent of n_1 and n_2. For convenience, let us first rename n_1 by n_1' and n_2 by n_1'', let p_1 be defined by the sequence of nodes $(n_1', \ldots, n_k')$ and let p_2 be defined by the sequence of nodes $(n_1'', \ldots, n_k'')$. We know that $\lambda_d(n_i') = \lambda_d(n_i'')$, for each $i = 1, k$.

Note that by applying R1 to saturation (after R3.i) all the pairs (n_i', n_i'') will be collapsed. By $f_{\text{R3i}}(d)$ we denote directly the result of R3i followed by these R1 steps. Let $n_{ii} \in \text{MBN}(f_{\text{R3i}}(d))$ denote the node that results from the collapsing of the (n_i', n_i'') pair.

First, if the two main branches n_1/p_1 and $n_1//p_2$ contain the output node, for any embedding e, $e(n_1')$ needs to be equal to $e(n_1'')$ and likewise, for each i the image of n_i' needs to be equal to the one of n_i''. The reason is that the two branches have the same endpoints (n_0 and $\text{OUT}(d)$), the same length and, since p_1 has no //-edge, n_0/p_1 needs to be isomorphic to the path from $e(n_0)$ to $e(\text{OUT}(d))$. Thus, it is obvious that by merging each n_i' and n_i'' we obtain a pattern that also has an embedding, if d does, and it maps $\text{OUT}(d)$ into the same node.

Let us now assume that p_2 ends above $\text{OUT}(d)$. Let m be the function from d into d that maps $\text{TP}_d(p_2)$ into $\text{SP}_d(n_2)$ and is the identity everywhere else (in particular, $m(p_2) = p_1$). We can show that $e \circ m$ is another embedding of d into t, one that takes each pair n_i', n_i'' into the same image and preserves the image of the output. The main reason is that p_1 and p_2 contain only /-edges, hence they can map only into a sequence of /-edges in t. And since the branches containing p_1 and p_2 respectively both start at n_1 and meet at $\text{OUT}(d)$ or at some other node above it, they have to map into the same path p of t, from $e(n_0)$ to $e(\text{OUT}(d))$. So $e(n_1')$ is either above or equal to $e(n_1'')$, because n_1' is connected by a /-edge to

its parent n_0). But then all the nodes on p_1 map above or in the same place in p as the nodes of p_2; in particular $e(n'_k)$ is above the image of any main branch node n that is //-child of n''_k. Therefore $e \circ m$ satisfies the condition imposed by the //-edge between n''_k and n. All the other conditions for showing $e \circ m$ is an embedding follow directly from the fact e is an embedding. The image of $\mathrm{OUT}(d)$ is the same in $e \circ m$ and e, since $\mathrm{OUT}(d)$ is not part of p_2.

We argue now that $e \circ m$ is also an embedding for the DAG d' obtained from d as follows: (a) for each i, append the predicate subtrees of n''_i below n'_i, (b) remove the edge n_0/n''_1 and the tree pattern $\mathrm{SP}_d(p_2)$, and (c) connect the dangling incoming //-edges of children of n''_k to n_k. (By the test conditions these must be the only dangling edges.)

But $f_{\mathrm{R3i}}(d)$ has a straightforward mapping h into d', as $h = \{n_{ii} \mapsto n'_i; x \mapsto x \text{ elsewhere }\}$, hence we obtain the desired embedding e' as $h \circ e \circ m$, with $e'(\mathrm{OUT}(f_{\mathrm{R3i}}(d))) = e(\mathrm{OUT}(d))$.

R3.ii: We can use the same idea as for R3.i.

R4.i: Let n_0 be the parent of n_1 and n_2. Suppose that d has an embedding e into a tree t. p_1 and p_2 are parts of main branches starting from n_0 and ending in a common node, at or above $\mathrm{OUT}(d)$. Hence, if d has an embedding e into a tree t, the nodes of p_1, of p_2 and each n_4 must all map into the same path p of t. Moreover, since n_0/p_1 has only /-edges, it is necessarily isomorphic to the fragment of p starting at n_0 and of length $|p_1| + 1$.

Let n_5 be the end node of p_1 and n_6 the end node of p_2. With necessity, either there is a node $n' \in p_1$ such that $e(n_6) = e(n')$ or $e(n_6)$ is below $e(n_5)$. In the former case, we can show that for any n_4 there is no node n'' of p_1 such that $e(n'') = e(n_4)$. For a given n_4, let us assume that such a node n'' exists. Since p_1 is isomorphic to $e(p_1)$, and the mapping of $p_2//n_4$ through e would imply also a mapping of $p_2//n_4$ into a suffix of p_1, this leads to a contradiction. Since n_0/p_1 is isomorphic to the beginning of p, it means that $e(n_4)$ is below $e(n_5)$. But then we can also map all nodes of $f_{\mathrm{R4i}}(d)$ exactly following e because e verifies the condition imposed by the //-edge between p_1 and n_4 and the part that was removed (p_2) was not connected to other main branches. Moreover, the image of the output is the

same, because $\mathrm{OUT}(d)$ is below p_1 and $f_{\mathrm{R4i}}(d)$ keeps all nodes under p_1 unchanged.

If $e(n_6)$ is below $e(n_5)$ in t, then, following the same reasoning, we can argue that e can be reused to map the nodes of $f_{\mathrm{R4i}}(d)$ into t.

R4.ii: Proof similar to R4.i: here the rule's test condition guarantees that, in any embedding, the beginning of p_1 is mapped below n_4.

R5: Let st denote the subtree predicate introduced by $f_{\mathrm{R5}}(d)$ on n_2, and let st' denote its copy under n_3. Given the embedding e of d in t, there are three possible cases:

- the image of n_3 is the same with the image of n_2,
- the image of n_3 is the same with the image of some other node n_4 from p_2,
- the image of n_3 is below the image of any node from p_2.

In the first case, e gives immediately an embedding e' of $f_{\mathrm{R5}}(d)$ in t, since st' has already an image in the subtree rooted at $e(n_3)$.

In the second case, we can first conclude that the DAG pattern $d' = \mathit{collapse}_d(n_4, n_3)$ must not be immediately unsatisfiable. This is because e gives also an embedding e'' from d' in t, one that maps the node $n_{3,4}$ into $e(n_3) = e(n_4)$. Let us now consider the tree pattern $p = \mathit{pattern}(\lambda_d(n_2)[Q])$, which modulo renaming is the pattern formed by the main branch node n_2 and the predicate subtree st. Since we know that p maps into $\mathrm{SP}_{d'}(n_2)$, by some mapping f, we can also build a containment mapping $\widetilde{f}$ from $f_{\mathrm{R5}}(d)$ into d', defined as follows: (a) n_3 and n_4 have the same image, $n_{3,4}$, (b) $\widetilde{f} = f(n)$ for all the nodes n of st and (c) $\widetilde{f}$ is the identity mapping for all the other nodes. Finally, we obtain the embedding e' as the composition $e'' \circ \widetilde{f}$.

In the third case, let us consider the DAG pattern d' obtained from d by appending the pattern of Q below $\mathrm{OUT}(\mathrm{TP}_d(p_2))$, via a //-edge (as described in the rule condition). Note that since the image of n_3 under e is below the image of any node from p_2, we can easily obtain from e an embedding e'' from d' into t.

Moreover, by the test condition, we have a root-mapping f from $\mathit{pattern}(\lambda_d(n_2)[Q])$ into the modified pattern, $\mathrm{TP}_{d'}(p_2)$. From f we will construct a mapping $\widetilde{f}$ from $f_{\mathrm{R5}}(d)$ in d'.

Let st be new subtree predicate in $f_{\mathrm{R5}}(d)$, corresponding to $[Q]$ at node n_2. We define $\widetilde{f}$ from $f_{\mathrm{R5}}(d)$ in d'as follows: (a) $\widetilde{f}$ is the identity function for nodes outside st and (b) $\widetilde{f}(n) = f(n)$ for all the nodes $n \in st$.

Finally, by the composition $e'' \circ \widetilde{f}$ we obtain an embedding of $f_{\mathrm{R5}}(d)$ into t.

R6: Let n_0 be the parent of n_1 and n_2. For convenience, let us first rename n_1 by n'_1 and n_2 by n''_1, let p_1 be defined by the sequence of nodes $(n'_1, \ldots, n'_k)$ and let p_2 be defined by the sequence of nodes $(n''_1, \ldots, n''_k)$. We know that $\lambda_d(n'_i) = \lambda_d(n''_i)$, for each $i = 1, k$.

As mentioned, our rule-rewriting algorithm would behave in the same way if R6 collapsed the entire p_1 and p_2 paths. To simplify the presentation of the rules, we delegated to R1 this task. However, to simplify the presentation of the proof, we will consider the pattern d'', representing the result of applying R6 followed by these R1 steps. Let $n_{ii} \in \mathrm{MBN}(d'')$ denote the node that results from the collapsing of the (n'_i, n''_i) pairs. It is easy to see that there is always a mapping h from $f_{\mathrm{R6}}(d)$ into d'', given by the composition of the applications of f_{R1} that merge all these (n'_i, n''_i) pairs. Hence, it is sufficient to show that for any embedding e of d into a tree t, there is an embedding e' of d'' into t, which guarantees that $h \circ e'$ is an embedding of $f_{\mathrm{R6}}(d)$ into t.

Let us consider any embedding e of d in a tree t. We have three possible cases:

- the image of n'_1 is the same with the image of n''_1,
- the image of n'_1 is above the image of n''_1,
- the image of n''_1 is above the image of n'_1.

In the first case, e gives immediately an embedding e' of d'' into t, since the nodes of p_1 and p_2 have the same images.

In the second case, let $e(p_1)$ denote the image of p_1 in t. We show that the function e' from d'' into t which: (a) for each i maps n_{ii} into $e(n'_i)$, and (b) maps n into $e(n)$ for all the nodes n outside $\mathrm{TP}_d(n_{11}/\ldots/n_{kk})$, can be extended to a full embedding of d'' into t. For that, we must show that all the predicate subtrees rooted at n_{ii} nodes can be mapped in the subtree rooted at $e(n'_i)$. Since predicate

subtrees from n'_i obviously have an image at $e(n'_i)$, e' is defined by e on these nodes. What remains is to describe e' over the predicates subtrees that originate at n''_i nodes.

Since $\mathrm{TP}_d(p_1)$ and $\mathrm{TP}_d(p_2)$ are similar, we always have a root-mapping from both /-patterns into any p_{12}, as given in Definition 3.3.1. Note that the embedding e (as any other embedding of d in general) imposes an order on the nodes of p_1 and p_2, consistent with their /-edges. We can thus always find an associated p_{12} /-pattern that has an embedding in the subtree rooted at $e(n'_1)$ in t. Let the /-pattern fixed in this way be p and let his embedding into the subtree rooted at $e(n'_1)$ be e''. Let f_2 be the root-mapping of $\mathrm{TP}_d(p_2)$ into p (by definition, we can always find such an f_2).

Finally, we can define e' over the predicates subtrees of n_{ii}'s that originate at n''_i's using the composition $e'' \circ f_2$.

Since predicate subtrees from n'_i obviously have an image at $e(n'_i)$, e' is defined by e on these nodes. Moreover, since $\mathrm{TP}_d(p_1)$ and $\mathrm{TP}_d(p_2)$ are similar, by definition it means that we can also map

By symmetry, the third case can be handled in the same manner.

R7: Let n_3 and n_4 denote the end points of p_2, let n_1 denote the common parent of p_1 and p_2 and let n_2 denote the image of n_3 under the mapping from $\mathrm{TP}_d(p_2)$ into $\mathrm{SP}_d(p_1)$.

Since n_3, n_4 and p_2 are not connected to any other parts of the DAG, for any embedding e into a tree t, the restrictions e' of e to $\mathrm{NODES}(d) \setminus \{n | n \in p_2\}$ is a partial embedding into t (because n_3, n_4, p_2 do not affect the conditions needed to embed the other nodes of d are still satisfied). But e' is a total embedding for $f_{\mathrm{R7}}(d)$ and $e'(\mathrm{OUT}(f_{\mathrm{R7}}(d))) = e'(\mathrm{OUT}(d)) = e(\mathrm{OUT}(d))$.

R8: Let n_0 denote the common parent of the two branches in parallel, and let n_3 denote their common child node. As the branches of p_1 and p_2 are fragments of main branches between the nodes n_0 and n_3, and $n_0/p_1/n_3$ has only /-edges, we can argue, as for R7, that for any embedding e into a tree t, p_1 and $e(p_1)$ are isomorphic. Also, $e(p_2) \subseteq e(p_1)$, as the nodes of $e(p_2)$ must lie on the same path fragment, between $e(n_0)$ and $e(n_3)$. Since $p_1 \to e(p_1)$ is an isomorphism, there is an

inverse mapping i from $e(p_1)$ into p_1 which is surjective. But $e(p_1) \supset e(p_2)$, hence $g = i \circ e$ is a mapping from p_2 into a suffix of p_1. As we assumed the conditions of R8 to be satisfied, it follows that $g(n_2) = n_1 \Leftrightarrow e(n_2) = e(n_1)$, for any embedding e. But then, if we take

$$e'(x) = \begin{cases} e(n_1)(= e(n_2)), \text{ if } x = n_{1,2} \\ e(x), \text{ otherwise} \end{cases}$$

e' will be an embedding for $f_{\mathrm{R8}}(d)$ with $e'(\mathrm{OUT}(f_{\mathrm{R8}}(d))) = e(\mathrm{OUT}(d))$.

■

Proof: (**Lemma 3.3.1**) Lemma 3.3.2 implies that, for every rule r and DAG d, $f_{\mathrm{r}}(d) \sqsubseteq d$ and 3.3.3 that $d \sqsubseteq f_{\mathrm{r}}(d)$. Hence $d \equiv f_{\mathrm{r}}(d)$. ■

Proof: (**Theorem 3.3.1**) By construction $d = dag(unfold(r))$ is such that it maps into $pattern(q)$, hence, by Lemma 3.2.1, $q' \sqsubseteq d$, where q' is q modified to have its output at the last main branch node of the prefix p.

By Lemma 3.3.1, every rule application preserves equivalence. Hence the final d is equivalent to the d initially built. Let r' be the rewriting returned at line 8. Since r' is just r with, possibly, some more navigation, and navigation is monotonic, then $d \sqsubseteq p$ guarantees that $unfold(r') \sqsubseteq q$. We already knew that d had a containment mapping into q'. Then, the last compensation just "moves" the output node lower, and guarantees that $q \sqsubseteq unfold(r')$. Hence $unfold(r') \equiv pattern(q)$. ■

Moreover, it is also complete, in the sense described in Section 3.2.

Theorem 3.3.2. *(1) Algorithm* Rewrite *is complete for rewriting XP . (2) If the input query q is minimal,* All-Rewrites *finds all minimal rewritings.*

Rewrite runs in worst-case exponential time as it uses a containment check (line 7) that is inherently hard:

Theorem 3.3.3. *Containment of a query $d \in XP^{\cap}$ into a query $p \in XP$ is coNP-complete in $|d|$ and $|p|$.*

Proof: To show that our problem is in coNP we can use an argument similar to the one used in [NS06] for the case of XPath with disjunction. We know that

$nf(d) = U$, where U is a union of queries from XP . A non-deterministic algorithm that decides $d \not\subseteq p$ guesses $u \in U$ (without computing U), making a certain choice at each step of interleaving. Then, it checks that $u \not\subseteq p$, which can be done in PTIME as $u, p \in XP$.

The hardness part is proven by reduction from the 3DNF-tautology problem [GJ79], which is known to be coNP-complete. We start from a 3DNF formula $\phi(\bar{x}) = C_1(\bar{x}) \vee C_2(\bar{x}) \vee \ldots C_m(\bar{x})$ over the boolean variables $\bar{x} = (x_1, \ldots x_n)$, where $C_i(\bar{x})$ are conjunctions of literals.

Out of ϕ we build $d \in XP^{\cap,\cup}$ and $p \in XP$ over $\Sigma = \{x_1, \ldots, x_n, b, v, true, false, yes\}$ such that ϕ is a tautology iff $d \subseteq p$.

Intuitively, d will encode all possible truth assignments for ϕ.

We build d by intersecting two branches, based on the following gadgets (illustrated in Figure 3.2):

1. the pattern $x_1[yes]//x_2[yes]//...//x_n[yes]$ (denoted T)
2. the pattern $x_1[true]/x_1[false]/x_2[true]/x_2[false]/\ldots/x_n[true]/x_n[false]$ (denoted S)
3. for each clause C_i, the pattern obtained from $x_1/a/x_1/a/x_2/a/\ldots/x_n/a/x_n/a$ by putting the $[yes]$ predicate below each of the 3 nodes corresponding to the literals of C_i (this pattern is denoted P_{C_i}). For instance, for $C_i = (x_1 \wedge \bar{x_2} \wedge x_3)$, we have the pattern $P_{C_i} = x_1/a[yes]/x_1/a/x_2/a/x_2/a[yes]/x_3/a[yes]/x_3/a\ldots/x_n/a/x_n/a$.
4. for each clause C_i, Q_{C_i} denotes the predicate $[c/c/c/\ldots/c/v[.//P_{C_i}]]$, with $m - i + 1$ c-nodes,
5. for each C_i, the predicate $Q_i = [Q_{C_1}, \ldots Q_{C_{i-1}}, Q_{C_{i+1}}, \ldots, Q_{C_m}]$, that is the list of all Q_{C_j} predicates for $j \neq i$.
6. the pattern $c[Q_1]/c[Q_2]/c[Q_3]/\ldots c[Q_m]/c$ (denoted U)

Now, we define d as

$$d = (doc(A)//T//out) \cap (doc(A)/U/\mathbf{v}/S/out).$$

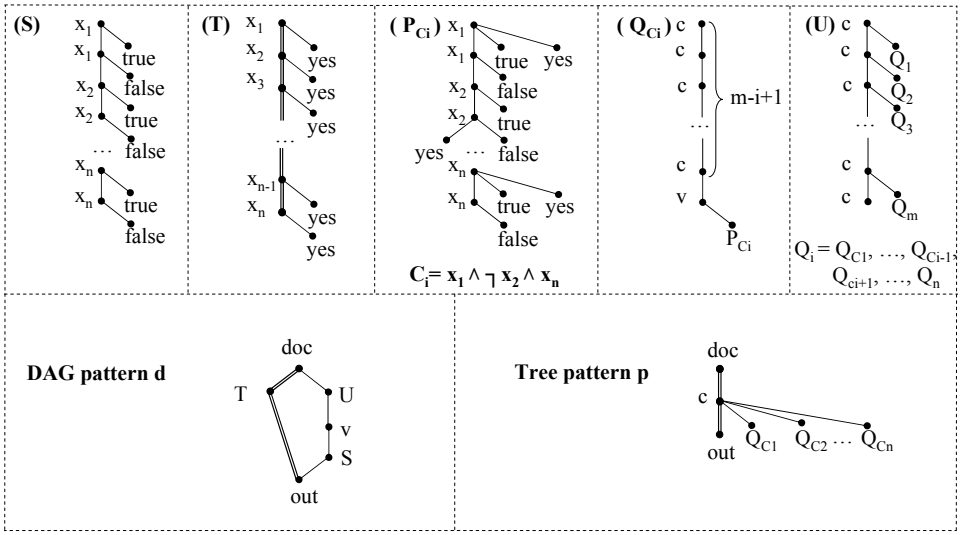


Figure 3.2: The patterns used in the reduction of TAUTOLOGY to DAG containment

An important observation is that Q_{C_i} predicates are not inherited at c-nodes. This is ensured by the v node we introduced in Q_{C_i}s and in U (for inheritance, the last c-node of Q_{C_i}'s branch would have to match with a v-node).

Next, when interleaving the two branches in d , x_k in the left branch can either be coalesced with the first x_k (having predicate $true$) in the second branch (having predicate $false$), corresponding to the case $x_k = \text{TRUE}$, or with the second x_k, corresponding to $x_k = \text{FALSE}$. Hence interleavings correspond to truth assignments for the variables of ϕ.

Note also that when some clause C_i is made TRUE by a truth assignment (i.e., all its literals are TRUE), then the predicate P_{C_i} will hold at the last c node in the U part, or, put otherwise, the Q_{C_i} predicate will now hold at the ith c node in U.

Finally, let p be the pattern

$$p = doc(a)//c[Q_{C_1}, Q_{C_2}, Q_{C_3}, \ldots, Q_{C_m}]//out.$$

We can now argue that $d \sqsubseteq p$ iff ϕ is a tautology. The if direction is immediate since in this case each truth assignment makes at least one C_i true.

This means that the Q_{C_i} predicate will now hold at the ith c-node in d and, since all other Q_{C_j} predicates, for $j \neq i$, were already explicitly present at this node, it is now easy to see that there exist now a containment mapping from p into d that takes p's c-node into d's ith c-node.

The only if direction is similar. If for some truth assignment, none of the clauses is TRUE (in the case ϕ is not a tautology), then it is easy to check that p will not have a containment mapping into the interleaving corresponding to that truth assignment. ■

One might hope there is an alternative polynomial time solution. We prove this is not the case.

Theorem 3.3.4. *Deciding the rewriting problem of a query q using a set of views $\mathcal{V}$ is coNP-complete.*

However, our rule rewriting procedure is polynomial:

Lemma 3.3.4. *The rewriting of a DAG d using* APPLY-RULES *always terminates, in* $\mathrm{O}(|\mathrm{NODES}(d)|^2)$ *steps.*

Proof: First, let us notice that none of the rules increases the number of main branch nodes, and in fact R1, R3, R4, R6, R7, R8 always decrease it, hence the number of times they are applied is less than $|\mathrm{MBN}(d)| = \mathrm{O}(|\mathrm{NODES}(d)|)$. R5 can also fire only a finite number of times, as the number of predicates to be introduced is bound by the initial number of predicates in the pattern d, which is in $\mathrm{O}(|\mathrm{NODES}(d)|)$.

R2 leaves the number of nodes unchanged and may decrease the number of edges by one or leave it the same. R2.i always progresses down main branches, and R2.ii always up, respectively. The only possibility of going into a loop would come from the interaction of R2.i and R2.ii.

Consider the generic case depicted in Figure 3.3(a), in which R2.ii would apply for nodes n_1, n_2, n_3. (The case in which we apply an R2.i step is symmetrical). Suppose that n_3 also has a /-edge towards a node n_4. There would be a danger of looping if R2.i had been previously applied to nodes n_3, n_2, n_4, and now it would apply again because $n_2//n_3$ would be re-introduced. But then, R2.i

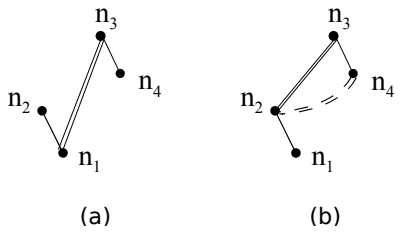


Figure 3.3: Interaction between R2.i and R2.ii

applied to those nodes would have introduced an edge $n_4//n_2$ and any later applications of R2 (or of any other rule) would have maintained n_2 reachable from n_3, as in Figure 3.3(b). In this case, R2.ii would not introduce any //-edge between n_3 and n_2, as it is explicitly specified in the rule.

Thus, R2 can fire at most $|\text{MBN}(d)|^2$ times, because at each step it infers the order, in all interleavings, of a pair of nodes from $\text{MBN}(d)$ whose ordering was unknown before. So, rewriting with R1-R8 always terminates in at most $O(|\text{NODES}(d)|^2)$ steps. ■

Corollary 3.3.1. EFFICIENT-RW *always runs in PTIME.*

PTIME Completeness. We consider next restrictions by which EFFICIENT-RW becomes also complete, thus turning into a complete and efficient rewriting algorithm. Note that one may impose restrictions on either the *XP* fragment used by the query and views, or on the rewrite plans that REWRITE deals with. We consider both cases, charting a tight tractability frontier for this problem.

Case 1: *XP* fragment for PTIME. By a *//-subpredicate st* we denote a predicate subtree whose root is connected by a //-edge to a /-path p that comes from the main branch node n to which st is associated (as in $n[\ldots[.//st]]$). p is called the *incoming /-path* of st and can be empty.

By *extended skeletons* (XP_{es}) we denote patterns having the following property: for any main branch node n and //-subpredicate st of n, there is no mapping (in either direction) between the code of the incoming /-path of st and the one of the /-path following n in the main branch (where the empty code is assumed to

map in any other code). Note that all the paths given in the running example are from this fragment. We can prove the following:

Theorem 3.3.5. *For any pattern d in $XP^{\cap}_{es}$, d is union-free iff the algorithm* Apply-Rules *rewrites d into a tree.*

Corollary 3.3.2 (XP_{es})**.** *Algorithm* Efficient-Rw *is complete for rewriting XP_{es}.*

Case 2: Rewrite-plans for PTIME. We also identify a large class of rewrite plans that lead to PTIME completeness. Let us first introduce the notion of */-tokens* of a tree pattern. More specifically, the main branch of a tree pattern p can be partitioned by its sub-sequences separated by //-edges, and each /-pattern from this partitioning is called a *token*. We can thus see a pattern p as a sequence of tokens (/-patterns) $p = t_1//t_2//\ldots//t_k$. We call t_1, the token starting with $\text{ROOT}(p)$, the *root token* of p. The token t_k, which ends by $\text{OUT}(p)$, is called the *result token* of p.

We say that two (or several) tree patterns are *akin* if their root tokens have the same main branch codes. For instance, while the views v_1 and v_2 from our example are not akin, v_1 is akin to:

$$v_2' : \mathsf{doc(\text{“}L\text{”})//figure[.//caption//label]//subfigure/image[ps]}.$$

In this setting, we can relax the syntactic restrictions and accept the class of patterns $XP_{//}$, obtained from extended skeletons by freely allowing //-edges in the predicates that are connected by a //-edge to the main branch (such as in v_2'). We can prove the following:

Theorem 3.3.6. *For DAGs of the form $d = \bigcap_j p_j$, where all p_j are in $XP_{//}$ and akin, d is union-free iff the algorithm* Apply-Rules *rewrites d into a tree.*

Corollary 3.3.3 ($XP_{//}$)**.** Efficient-Rw *always finds a rewriting for $XP_{//}$, provided there is at least a rewriting r such that the patterns intersected in unfold(r) are akin.*

Tractability Frontier. We show next that relaxing any of these restrictions leads to hardness for rewriting and union-freedom:

Theorem 3.3.7. *(1) For a pattern d in $XP^{\cap}_{//}$, deciding if d is union-free is coNP-complete. (2) For a pattern $d = \bigcap_j p_j$, where all p_j are in XP and akin, deciding if d is union-free is coNP-hard.*

Please note that the rewriting problem can be solved using an oracle for union-freedom, but this does not provide any easy map reduction. This is why we prove the following result independently:

Theorem 3.3.8. *(1) Deciding the existence of a rewriting for a query and views from $XP_{//}$ is coNP-complete. (2) For a query and views from XP, deciding the existence of a rewriting r such that the patterns intersected in unfold(r) are akin is coNP-complete.*

Proofs for theorems 3.3.2, 3.3.5, 3.3.6 are in Appendix 3.A. The proofs of theorems 3.3.7 and 3.3.8 are in Appendix 3.B. Theorem 3.3.4 is a direct consequence of Theorem 3.3.7 (or alternatively of Theorem 3.3.8).

3.4 Nested Intersection

In the following, we will consider an extension *XPint* of *XP* with respect to intersection, which includes $XP^{\cap}$. The grammar of *XPint* is obtained from that of *XP* by adding the rule:

$$\begin{aligned} jpath \;::=\; & apath \mid (jpath) \mid jpath \cap jpath \mid \\ & (jpath)/rpath \mid (jpath)//rpath \end{aligned}$$

The semantics is similar to the one given in Definition 3.2.4, to which we add the one for intersection:

$$[\![rpath_1 \cap rpath_2]\!]_t = \{(n, n') | (n, n') \in [\![rpath_1]\!]_t \cap [\![rpath_2]\!]_t\}$$

Let us notice that *XPint* queries can also be represented by DAG patterns having the property that if there are two distinct main branches from a node n_1 to another node n_2, then n_1 must be the root of the DAG.

We start by extending the definition of an unfolding. Given a query $r \in$ *XPint* over the view documents $D_{\mathcal{V}}$, we define *unfold*(r) as the *XPint* query obtained by replacing in r each *doc*("V")$/V$ with the definition of V.

The notion of rewriting also extends naturally.

Definition 3.4.1. *For a given XML document D, an XPint query q and XPint views $\mathcal{V}$ over D, an XPint* -rewrite plan *of q using $\mathcal{V}$ is a query $r \in$ XPint over $D_{\mathcal{V}}$.*

If unfold$(r) \equiv q$, then we also say r is an XPint -rewriting.

Lemma 3.4.1. *An XPint -rewrite plan can be evaluated over a set of view documents $D_{\mathcal{V}}$ in polynomial time in the size of $D_{\mathcal{V}}$.*

The interesting result is that the rewriting problem stays in the same complexity class, even if the expressivity of the rewriting language increases:

Theorem 3.4.1. *Deciding the existence of an XPint -rewriting for a query and views from $XP_{//}$ is coNP-complete.*

Proof: It is similar to the proof of Theorem 3.3.8. ■

3.4.1 Rewritings Having Nested Intersection

We introduce the notion of *graph pattern* which is similar to a DAG pattern, but

- does not have a ROOT,
- it has *view nodes* with labels of the form *doc*(V) where V is a symbol from a set of views $\mathcal{V}$

In the same way build a DAG †q for a query $q \in XP^{\cap}$, we also build a graph pattern `graph`(q') for a query $q \in$ *XPint* . We omit the details.

We also generalize the function *unfold*(r) to operate on graphs. *unfold*() returns the *XPint* query that corresponds to replacing every *doc*$(V)/V$ with the query that defines V.

We can easily extend Theorem 3.2.1 to *XPint* :

Theorem 3.4.2. *For any $q \in XPint$ and any tree t, $q(t) = \texttt{graph}(q)(t)$.*

Starting from the pattern of the input query q and the views $\mathcal{V}$ we build, step by step, a graph $cand_{RW}(q, \mathcal{V})$ that we call the rewriting candidate. We then prove that the obtained candidate is *minimally containing* w.r.t. q, i.e. $cand_{RW}(q, \mathcal{V})$ is contained in any query that contains q. This guarantees completeness in the sense that if an *XPint* rewriting exists, $xpath(cand_{RW}(q, \mathcal{V}))$ is also a rewriting.

For a DAG pattern p, we designate by $\cup_i p_i = p$ the union of all its interleavings.

Algorithm BuildRewriteCandidate
input: query $q \in XPint$ with $p = pattern(q)$, set of views $\mathcal{V}$ defined by *XPint* queries
output: candidate rewriting $cand_{RW}(q, \mathcal{V})$

1. set r to p
2. let $S \leftarrow \{(V_i, o_{i,k}) \mid$ if $d_i = dag(unfold(V_i)) = \cup_j d_{ij}$ and $p = \cup_t p_t$,
   there is a mapping h_{ij} from every d_{ij} into some p_t,
   $o_{i,k} = h_j(\mathrm{OUT}(d_{ij})), \forall j\}$
3. foreach (V, o) in S
   add to r a new view-node $n_{V,o}$ labeled $doc(V)$ and an edge labeled V from $n_{V,o}$ to o
4. keep in r only paths accessible starting from view-nodes $V \in \mathcal{V}$
5. if $\mathrm{OUT}(p)$ is not in r, then **fail**
6. set $\mathrm{OUT}(r)$ to $\mathrm{OUT}(p)$
7. **return** $xpath(r)$ `▷ (this is $cand_{RW}(q, \mathcal{V})$)

Algorithm NestedRewrite

1. $cand_{RW}(q, \mathcal{V}) \leftarrow$ **BuildRewriteCandidate**$(q, \mathcal{V})$

2. if $cand_{RW}(q, \mathcal{V}) \equiv q$ then output $cand_{RW}(q, \mathcal{V})$
   else **fail**

The equivalence test at step (2) uses Lemma 3.2.3 and of the following result:

Lemma 3.4.2. *Let $p = \cup_i p_i$ and $q = \cup_j q_j$ be two (finite) unions of tree patterns. Then $p \sqsubseteq q$ iff $\forall i, \exists j$ s.t. $p_i \sqsubseteq q_j$.*

Proof: The proof is similar to the one for checking containment of unions of conjunctive queries [SY80].

The *if* direction is trivial. For the *only if* part, let p_i be one of the interleavings of p. Build a model m_p for p as in Lemma 3.2.2, by replacing each //-edge with two /-edges separated by a node having labeled z, where z is a fresh new label. Let n' be the node in m_p corresponding to $\text{OUT}(p)$ in p. Since $p \subseteq q$, there must be an interleaving q_j of q such that $n' \in q_j(m_p)$. But this implies the existence of a mapping from q_j into p_i, as q_j has no z labeled nodes. ■

Thus there is a simple (albeit EXPTIME) way to test containment or equivalence between two graphs: write the two graphs as the unions of their interleavings and then use Lemma 3.4.2 to check containment.

What we can guarantee for our algorithm is that it always produces the minimally containing rewriting, in the following sense.

Lemma 3.4.3. *For $q \in$ XPint , $\mathcal{V}$ a set of XPint views and r' a graph, if $q \sqsubseteq unfold(r')$, then $unfold(cand_{RW}(q, \mathcal{V})) \sqsubseteq unfold(r')$.*

Proof: Let d be $dag(unfold(cand_{RW}(q, \mathcal{V})))$, d' be $dag(unfold(r'))$ and p be $dag(q)$. As $q \sqsubseteq unfold(r')$. We can write $d = \cup_i d_i$, $d' = \cup_j d'_j$ and $p = \cup_k p_k$ as the union of their interleavings. By Lemma 3.4.2, there are containment mappings h_i from each d_i into some p_k and h'_j from every d'_j into some p_k.

We will prove by structural induction that there is a containment mapping from every d'_j into some d_i.

(1) Suppose r' is of the form $doc(V)/V$, where $V \in \mathcal{V}$. Since every d'_j maps into some p_k and $cand_{RW}(q, \mathcal{V})$ is built using all possible mappings of the views,

there is at least one occurence of V in d connected to a node o. Moreover, since $q \sqsubseteq \mathit{unfold}(r')$, there must be such an occurrence in which o actually corresponds to $\mathrm{OUT}(p)$. Hence there is trivially a containment mapping from every d'_j into some d_i.

(2) Consider now the case in which r' corresponds to a query of the form α/x, where every interleaving α_j of the pattern $\mathit{unfold}(\alpha)$ has a mapping m_j^1 into some d_i and x is a relative path. Let n_α be the last node on the main branches of $\mathit{unfold}(\alpha)$. Let d'_j be α_j/x. By construction, the subgraph of d_i accessible starting from $m_j^1(n_\alpha)$, call it t_{ij} is a tree that is isomorphic to a subtree st of $h'_j(d'_j)$, where h'_j is the mapping from d'_j into some interleaving p_k. Hence the identity function id is a mapping from st into t_{ij}, therefore m_j^1 can be extended to a mapping m_j^2 from d'_j into d_i. If $\mathrm{OUT}(d')$ is in the pattern of α, then from the containment of q into α/x we can also infer the containment of q into α and the induction hypothesis guarantees that m_1 can be chosen such that it is a containment mapping. Otherwise, $\mathrm{OUT}(d')$ is part of the pattern of x and $h'_j(\mathrm{OUT}(d')) = \mathrm{OUT}(p)$. But then $m_j^2(\mathrm{OUT}(d')) = h'_j(id(\mathrm{OUT}(d'))) = \mathrm{OUT}(d)$ and m_j^2 is a containment mapping.

(3) Suppose that r' is of the form $(\alpha \cap \mathit{doc}(V)/V)$, where every interleaving α_j of the pattern of $\mathit{unfold}(\alpha)$ has a mapping m_j^1 into some d_i. Let n_α be the last node in the main branches of $\mathit{dag}(\mathit{unfold}(\alpha))$ and $n'_\alpha = m_j^1(n_\alpha)$, unique for all j. Then again we can find $\mathit{doc}(V)/V$ in d that was added to $\mathit{cand}_{RW}(q, \mathcal{V})$ by a set of mappings $\{m'_l\}$ into p that agree with h'_j on $\mathit{unfold}(\mathit{doc}(V)/V)$ and such that $m'_l(n'_\alpha) = \mathrm{OUT}(p)$, $\forall l$. Hence n'_α is the output node of d and m_j^1 can be extended to a mapping m_j^2 from d'_j into d_i such that $m_j^2(\mathrm{OUT}(d')) = n'_\alpha = \mathrm{OUT}(d)$.

Then by Lemma 3.2.1, $d \sqsubseteq d'$. ■

Theorem 3.4.3. *For any query $q \in$ XPint and any XPint rewriting r' such that $\mathit{unfold}(r') \sqsupseteq q$, $\mathit{unfold}(\mathit{cand}_{RW}(q, \mathcal{V})) \sqsubseteq \mathit{unfold}(r')$.*

Proof: Follows from Theorem 3.4.2 and Lemma 3.4.3. ■

The soundness and completeness of the algorithm come as a corollary:

Corollary 3.4.1. *For any query $q \in$ XPint and set of views $\mathcal{V} \subset$ XPint , if q has an XPint rewriting using $\mathcal{V}$, then $\mathit{cand}_{RW}(q, \mathcal{V})$ is a rewriting.*

Running time The algorithm BuildRewriteCandidate runs in PTIME in the size of the query and of the views. However, NestedRewrite is exponential in the worst case, which is the best we can hope for, given the hardness result of Theorem 3.4.1.

3.5 Related Work

XPath rewriting using only one view (no intersection) was the target of several studies [XÖ05, MS05, TZ05, YLH03]. Previously proposed join-based rewriting methods either give no completeness guarantees [BÖB+04, TYO+08] or can do so only if the query engine has extra knowledge about the structure and nesting depth of the XML document [ABMP07]. Others [TYO+08] can only be used if the node ids are in a special encoding, containing structural information. Our algorithm works for any documents and type of identifiers, including application level ids, such as the id attributes defined in the XML standard [BPSM+06] or XML Schema keys [FW04].

In [LWZ06] and [GWY07], the authors look at a different problem, that of finding maximally contained rewritings of XPath queries using views. Rewriting more expressive XML queries using views was studied in [CR02, DT03a, ODPC06], but without considering intersection. Fan et al [FGJK07] define views using DTDs instead of queries and study the problem of rewriting an XPath using one view DTD.

Containment and satisfiability for several extensions of XPath with intersection have been previously investigated, but all considered problems were at least NP-hard or coNP-hard. For our language, containment is also intractable, but the equivalence test used in the rewriting algorithm is in PTIME for practically relevant restrictions. Satisfiability of XPath in the presence of the intersect operator and of wildcards was analyzed in [Hid03], which proved its NP-completeness. As noticed in [BFG05], there is a tight relationship between satisfiability and containment for languages that can express unsatisfiable queries. If containment is in the class K, satisfiability is in coK and if satisfiability is K-hard, containment is

coK-hard. We give even stronger coNP completeness results for the containment of an XPath p_1 into an XPath p_2, by allowing intersection only in p_1 and disallowing wildcards. Satisfiability is analyzed in [BFG05] for various fragments of XPath, including negation and disjunction, which could together simulate intersection, but lead to coPSPACE-hardness for checking containment. Richer sublanguages of XPath 2.0, including path intersection and equality, are considered in [tCL07], where complexity of checking containment goes up to EXPTIME or higher. None of these studies tries to identify an efficient test for using intersection in query rewriting. A different approach, taken by [GBG06] is to replace intersection by using a rich set of language features, and then try to simplify the expression using heuristics.

Finally, closure under intersection was analyzed in [BFK05] for various XPath fragments, all of which use wildcard. We study the case without wildcard and prove that *union-freedom* (equivalence between an intersection of XPaths and an XPath without intersection or union) is coNP-hard. However, under restrictions similar to those for the rewriting problem, union-freedom can be solved in polynomial time. Thus, we also answer a question we previously raised in [CAM07] regarding whether an intersection of XPath queries without wildcard can be reduced in PTIME to only one XPath.

APPENDIX TO THE CHAPTER

3.A Completeness Proofs

Proof: (**Theorem 3.3.2(1)**) If there is a DAG d equivalent to the query q, then $q \sqsubseteq d$, and then, by Lemma 3.2.2, there must be a containment mapping from d into q. Thus, if there is a rewriting of the form $\mathcal{I}$ or $\mathcal{I}/comp$ or $\mathcal{I}//comp$, where $\mathcal{I}$ is an intersection, in particular there must be a root-mapping from $unfold(\mathcal{I})$ into q. Then, looking for all rewrite plans amounts to testing for all prefixes of $\mathrm{MB}(q)$ if there is an intersection of views, with possibly some compensation on each branch, that map their main branches into that prefix, then add the compensation below

the prefix, which will guarantee a containment mapping from the unfolding into the entire q.

Step 1 in REWRITE tries all prefixes of $\mathrm{MB}(q)$ and for each such prefix it finds all views that map their main branch inside that prefix. Hence, for each $\mathcal{I}$ as before, there will be a $\mathcal{I}'$, built at step 4, of the form $\mathcal{I} \cap \mathcal{J}$, where $\mathcal{J}$, possibly empty, is an intersection of other views (with maybe compensation) that map their main branches inside the same prefix. But then $q \sqsubseteq \mathcal{I}' \sqsubseteq \mathcal{I}$, so $\mathcal{I}$ is a rewriting iff $\mathcal{I}'$ is one.

We argue next that in order to test whether $r' = \mathrm{compensate}(r, q, \mathrm{OUT}(p))$ is an equivalent rewrite plan (which amounts here to testing that $\mathit{unfold}(r') \sqsubseteq q$) it is sufficient (and obviously necessary) to test that $d \sqsubseteq p$ (in REWRITE at line 7). In other words, if the test $d \sqsubseteq p$ fails, then r (potentially compensated) cannot yield an equivalent rewriting.

The test for $d \sqsubseteq p$ amounts to (a) testing that d is union-free, with some interleaving i such that $i \equiv d$, and (b) testing that $i \sqsubseteq p$. If d is union-free, then the statement $\mathit{unfold}(r') \sqsubseteq q$ *iff* $d \sqsubseteq p$ can be proven in straightforward manner.

We will now show that $d' = \mathit{unfold}(r')$ cannot be union-free if d is not union-free.

Claim 3.A.1. *d' is union-free only if d is union-free.*

Proof of the claim. Let $d \equiv i_1 \cup \ldots i_k$, for $i_1, \ldots, i_k$ being the non-reducible interleavings of d (for each i_j we cannot build *another*, i.e. non-equivalent, interleaving that contains i_j). For each i_j, let i'_j denote the compensated interleaving

$$i'_j = \mathrm{compensate}(i_j, q, \mathrm{OUT}(p)).$$

First, it is straightforward to show that

$$d' \equiv \bigcup_j i'_j$$

by showing containment mappings in both directions.

We show next that if d is not union-free, then $\bigcup_j i'_j$ is not union-free either (there is no query in this union that contains all other queries). We will rely on the following two observations:

1. The compensation applied on d's interleavings will only extend the main branch, but will not bring (or qualify) new predicates for the existing main branch nodes, hence yielding no new mapping opportunities in this sense. This is because the node $\text{OUT}(p)$ in prefix p is assumed to have already "inherited" the rest of q as a predicate (p is a lossless prefix).

   In other words, if there exists a containment mapping between i'_j and i'_l, then there must exist a root-mapping between the corresponding i_j and i_l.

2. We can partition the interleavings of d into two classes:

   (a) those that have a "minimal" (or certain) result token (essentially obtained after applying R1 rewrite steps on the result tokens of d's parallel branches),

   (b) the remaining ones, which by definition must have a result token that cannot map in the result token of interleavings of the first kind (the result token has a longer main branch or some predicates that are not necessarily present in all interleavings[1]).

   The first class cannot be empty, the second one may be empty.

By the first observation, for any two interleavings of d, i_j and i_l, knowing that $i_j \not\sqsubseteq i_l$, we can have $i'_j \sqsubseteq i'_l$ only if there exists already a root-mapping ϕ from i_l into i_j which fails to be a full containment mapping only because the image of $\text{OUT}(i_l)$ is not $\text{OUT}(i_j)$. If such an root-mapping does not exist then surely we have that $i'_j \not\sqsubseteq i'_l$.

By the second observation, for interleavings of the first kind, their result token will always map in the result token of any other interleaving (of both kinds) such that the image of the output node is the output node in the other interleaving.

Putting everything together, since the first class cannot be empty, in order to prove the claim it is now sufficient to show that a query i'_l corresponding to an interleaving i_l of the second kind cannot reduce (contain) a query i'_j corresponding

[1]Since this is the result token and everything else can be put "above" it, it is easy to obtain this minimal, certain result token.

to an interleaving i_j of the first kind. Let i_j and i_l be the two interleavings, with t_j^o and t_l^o denoting their result tokens, such that there exists a root-mapping ϕ which takes $\text{OUT}(i_l)$ into some main branch node that is not part of t_j^o (by the definition of the two kinds of interleavings t_l^o cannot map into t_j^o).

We will show that this leads to a contradiction, namely that i_j *is not minimal*, in the sense that a main branch node of i_j can be removed, obtaining another valid interleaving of d which contains i_j but is not equivalent to it.

Since all the branches in parallel in d (denoted hereafter x) must have a containment mapping ψ_x into i_l (and a containment mapping τ_x into i_j as well), we obtain by $\phi \circ \psi_x$ a root-mapping from each branch x into i_j. Importantly, the image of $\text{OUT}(x)$, $n = \phi(\psi_x(\text{OUT}(x)))$, is some main branch node of i_j *above* t_j^o.

This means that we can also map (by a root-mapping) each x branch into i_j somewhere higher than the result token t_j^o. But this hints that we can in fact simplify i_j into an interleaving having a shorter main branch, yet containing i_j. More precisely, this interleaving of the parallel branches x, described by a code-mapping pair (i, f_i), can be obtained as follows:

1. take as the code i the main branch of i_j *without* n,

2. define f_i as (a) for the main branch nodes of each x except those of the result token, by the $\phi \circ \psi_x$ mapping, (b) for the main branch nodes of the result token of x by τ_x.

It is easy to check that the interleaving obtained in this way has a containment mapping into i_j based on the "identity" mapping for the main branch nodes. **End of the claim**

Finally, since the algorithm is sound (Theorem 3.3.1), any rewrite plan that is output is indeed a rewriting.

Observation for the restricted PTIME fragments. Note that for a given a query q in XP_{es}or $XP_{//}$, its lossless prefixes may not remain in the fragment. This can be easily be avoided in order to enable the same Rewrite

execution without additional changes, by adapting the definition of lossless prefix as follows:

1. for the fragment XP_{es}: (a) if q is of the form $p/t//r$ (where t is not empty and r may be empty), add only as side branch on OUT(p) the t subpattern, (b) if q is of the form $p//r$ (for r not empty), no additional predicates need to be added on OUT(p).

2. for the fragment $XP_{//}$: (a) if q is of the form $p/t//r$ (where t is not empty and r may be empty), add as side branches on OUT(p) the t subpattern and the //t//r subpattern, (b) if q is of the form $p//r$ (for r not empty), add as side branch the $//r$ subpattern.

It is easy to check that with this adjustment the lossless prefixes remain in the corresponding fragment and, importantly, the necessary property *"if there exists a containment mapping between i'_j and i'_l, then there must exist a root-mapping between the corresponding i_j and i_l"* remains true (no new predicate matching opportunities are brought by the compensation step). ■

Proof: (**Theorem 3.3.2(2)**) If q is minimal, all the subtrees added by the function compensate are also minimal, hence the rewritings that are produced are minimal. It is left to prove that All-Rewrites finds them all. We already saw that if there is a rewriting, Rewrite tries one which is equivalent to it. All-Rewrites does even more: it really tries all the rewrite plans, because it considers all subsets of views that map down to a certain position in MB(q), at line $2'$. Any other compensation *comp* that may be needed is copied from q in the call to compensate from the return clause (line 8). It is easy to prove that a tree pattern is minimal iff one cannot drop subtrees from it (which is the definition of minimality in [MS05]). Therefore, if q is minimal, all its subtrees are also minimal. ■

3.A.1 Cases Leading to PTIME Completeness

For the polynomial time cases, we remind some of the definitions from Section 3.3.2 and introduce a few new ones.

A *//-predicate* is a predicate subtree connected by a //-edge to the main branch.

A *tree skeleton* is a tree pattern without //-edges in predicate subtrees.

Given a pattern d, by its *extended skeleton* we denote the XP_{es} pattern $s(d)$ obtained from p by pruning out the //-predicates that do not obey the condition for XP_{es} .

In the following, for simplicity, by the *skeleton* of a tree or DAG pattern we denote its *extended skeleton*. Unless stated otherwise, all the patterns are from $XP_{//}$.

For a pattern p, we call t_1, the token starting with $\mathrm{ROOT}(p)$, the *root token* of p. The token t_k, which ends by $\mathrm{OUT}(p)$, is called the *result token* of p. The other tokens are denoted *intermediary tokens*, and by the *intermediary part* of a tree pattern we denote the sequence of intermediary tokens. Note that a tree pattern may have only one token, if it does not have //-edges in the main branch. By a *token-suffix* of p we denote any tree pattern defined by a suffix of the sequence of tokens $(t_1, \ldots, t_k)$. Symmetrically, we introduce the notion of *token-prefix* of p.

Lemma 3.A.1. *Two patterns in XP are equivalent iff they are isomorphic after minimization.*

Proof: It is a direct consequence of Theorem 1 from [MS05], because equivalence in XP (we remind that our language XP has no wildcard) is always witnessed by containment mappings in both directions. ■

Lemma 3.A.2. *A DAG pattern d is union-free only if its skeleton DAG pattern, $s(d)$, is union-free.*

Proof: The proof is based on the following property: modulo //-predicates, the sets of tree patterns $interleave(d)$ and $interleave(s(d))$ are the same. More precisely, for each $p_i \in interleave(d)$ there exists $p'_i \in interleave(s(d))$ such that $s(p_i) = p'_i$ and the other way round. Supposing that $s(d)$ is not union-free, let us assume towards a contradiction that d is union-free. Let p_i denote the interleaving such that $p_i \equiv d$ and let p'_i denote the associated interleaving from $s(d)$, $p'_i = s(p_i)$. Since $s(d)$ is not union-free, there exists some $p'_j \in interleave(s(d))$ such that

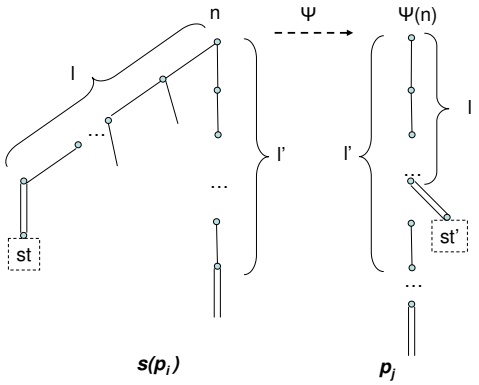


Figure 3.4: Skeletons and mappings.

$p'_j \not\sqsubseteq p'_i$. Then there is $p_j \in \mathit{interleave}(d)$ such that $s(p_j) \not\sqsubseteq s(p_i)$. Finally, since $p_j \sqsubseteq p_i$ we also have $p_j \sqsubseteq s(p_i)$.

Suppose for the sake of contradiction that $p_j \sqsubseteq s(p_i)$ and $s(p_j) \not\sqsubseteq s(p_i)$ both hold. Then any containment mapping ψ from $s(p_i)$ into p_j should use some of the predicates of p_j starting with a //-edge (otherwise we would have a containment mapping from $s(p_i)$ into $s(p_j)$ as well). But this is not possible since for any //-subpredicate st, its incoming /-path l is incompatible (does not map) with the path l' following the main branch node (see Figure 3.4). ■

Lemma 3.A.3. *Given two skeleton patterns v_1, v_2 such that their root and result tokens have the same main branch, the DAG pattern $d = dag(v_1 \cap v_2)$ is union-free iff Algorithm* Apply-Rules *rewrites d into a tree.*

Proof: Let us first consider what rule steps may apply in order to refine d. First, since we are dealing with patterns with root and result tokens having the same main branch, R1 steps will first apply, coalescing the root and result tokens of the two branches. At this point, the only rules that remain applicable are R6 and R7. This is because we don't have nodes with incoming (or outgoing) /-edges and //-edges simultaneously and R5 will only apply to predicates starting by a //-edge.

We continue by the following claim:

Claim 3.A.2. *d is union-free iff rule R7 applies.*

Note that since we only have 2 parallel branches an application of rule R7 would immediately yield a tree. So the *if* direction is straightforward. For the *only if* direction, if R7 does not apply this translates into

(†) *there is no mapping (not necessarily root-mapping) between the intermediary part of* v_1 *and the intermediary part of* v_2*.*

Assuming that (†) holds, rule R6 remains the only option. So, possibly after some applications of R6, followed by applications of R1 collapsing entire tokens, we obtain a refined DAG d as illustrated in Figure 3.5 (only the main branches of d are illustrated). p_r has the common main branch following the root (may have several tokens if R6 applied) and t_o denotes the result token. t_1 and t_2 denote the two sibling /-patterns for which R6 no longer applies. Since t_1 and t_2 are *dissimilar* we have that $t_1 \not\equiv t_2$.

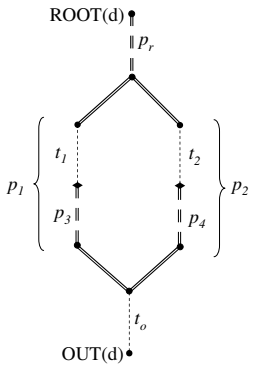


Figure 3.5: DAG pattern d after (possibly) applying R6 and R1.

We show that d is not union-free by the following approach: we build two interleavings, p' and p'', that do not contain one another, and then show that by assuming the existence of a third interleaving p that contains both we obtain the contradiction $t_1 \equiv t_2$.

We will use the following claim:

Claim 3.A.3. *Given two /-patterns* t_1 *and* t_2 *from* XP_{es}*, if* t_1 *does not map in* t_2 *then, for any tree pattern* q *of the form* $\ldots//t_2'//\ldots$ *with* t_2' *being an isomorphic*

copy of t_2, we have that t_1 does not map into t'_2

Proof: Follows easily from the restriction on // usage in predicates. ■

The following steps will implicitly use the claim above.

Let $p_1 = t_1//p_3$ denote the left branch and let $p_2 = t_2//p_4$ denote the right branch in d. Because of (†), there is no mapping (not necessarily root-mapping) between p_1 and p_2.

Let sf_1 denote the maximal token-suffix of p_1 that can map into p_2, and let pr_1 denote the remaining part (i.e., a token-prefix). Note that pr_1 cannot be empty. So we can write p_1 as $p_1 = pr_1//sf_1$.

Similarly, let sf_2 denote the maximal token-suffix of p_2 that can map into p_1, and let pr_2 denote the remaining part, non-empty as well. So we can write p_2 as $p_2 = pr_2//sf_2$.

We build p' as

$$p' = p'_r//pr'_2//p'_1//t'_o = p'_r//pr'_2//pr'_1//sf'_1//t'_o$$

and p'' as

$$p'' = p''_r//pr''_1//p''_2//t''_o = p''_r//pr''_1//pr''_2//sf''_2//t''_o.$$

where the $\#'$, $\#''$ parts are isomorphic copies of the $\#$ parts of d.

Note that pr'_1 (resp. pr''_2) starts by token $t'_1 \equiv t_1$ (resp. $t''_2 \equiv t_2$).

These two queries are obviously in $interleave(d)$. Moreover, there can be no containment mapping between p' and p'' since, by the way sf_1 and sf_2 were defined, pr'_1 (resp. pr''_2) could only map in pr''_1 (resp. pr'_2).

So neither p' nor p'' can be the interleaving that reduces all the others. We show in the following that no other interleaving p of d can reduce both p' and p'' unless $t_1 \equiv t_2$.

Let us assume that such a p exists. Without loss of generality, let p be of the form

$$p = p_r//m//t_o.$$

(interleavings that are not of this kind will not remain in the normal form of d).

We assume a containment mapping ϕ' from p into p' and a containment mapping ϕ'' from p into p''. Obviously, v_1, v_2 must have containment mappings

into p, since $p \equiv v_1 \cap v_2$. In particular, their sub-sequences p_1 and p_2 have images in the m part of p. Let ψ' and ψ'' be these containment mappings.

With a slight abuse of notation, let $\psi'(pr_1)$ denote the minimal token-prefix of m within which the image under ψ' of the pr_1 part of v_1 occurs. $\psi'(pr_1)$ is well defined because v_1 and v_2, and hence p_1 and p_2, are in XP_{es}, hence the image of a token of p_1 and of its predicates is included into a token of m. In other words, $\psi'(pr_1)$ starts with the first token of m and ends with the token into which the last token of pr_1 maps. Similarly, let $\psi''(pr_2)$ denote the minimal token-prefix of m within which the image under ψ'' of the pr_2 part of v_2 occurs.

We can thus write p as

$$p = p_r//\psi'(pr_1)\ldots\psi'(sf_1)//t_o$$

and also as

$$p = p_r//\psi''(pr_2)\ldots\psi'(sf_2)//t_o.$$

Next, we argue that in the containment mapping ϕ'' of p in p'', we must have $\phi''(\psi'(pr_1)) = pr_1''$. Similarly, we must have that $\phi'(\psi''(pr_2)) = pr_2'$. This follows easily from the way sf_1 and sf_2 were defined. (For instance, no node of $\psi'(pr_1)$ can map below pr_2'' in p'', otherwise sf_1 would not be maximal. And $\text{MB}(p_r) = \text{MB}(p_r'')$ and $|\psi'(pr_1) \geq pr_1''|$, hence no node of $\psi'(pr_1)$ can map higher than pr_1'' either, otherwise p_r would not map into p''.) And it then implies that $\psi'(pr_1) \equiv pr_1$ and $\psi''(pr_2) \equiv pr_2$.

But since m starts by both the token-prefix $\psi'(pr_1)$ and by $\psi''(pr_2)$, hence by token-prefixes pr_1 and pr_2, it means that pr_1 and pr_2 should at least start by the same token. Hence $t_1 \equiv t_2$, which is a contradiction.

Thus, we proved that d is union-free iff, after a certain sequence of rule application, R7 applies, transforming the pattern into a tree. As we know from Lemma 3.3.4 that Apply-Rules also terminates, it follows that d is union-free iff Apply-Rules rewrites d into a tree. ■

For $d = v_1 \cap v_2$, where v_1 and v_2 are two skeleton patterns such that their root and result tokens have the same main branch, with the previous notations, we can also easily prove the following claim:

Claim 3.A.4. *All the interleavings of nf(d) are of the form*

$$p_r//\ldots//t_o.$$

In conclusion, we know for now that the intersection d of two skeleton queries such that their root and result tokens have the same main branch is union-free iff Apply-Rules rewrites d into a tree. Moreover, this happens iff there is a mapping between the intermediary part of one into the intermediary part of the other. Then, if d is not union-free, the result is a union of queries having the same root and result tokens, as described in Claim 3.A.4.

Lemma 3.A.4. *Given n skeleton patterns $v_1, \ldots, v_n$ such that their root and result tokens have the same main branch, the DAG pattern $d = dag(v_1 \cap \cdots \cap v_n)$ is union-free iff Algorithm* Apply-Rules *rewrites d into a tree. If the skeletons are of the form $v_i = p_r//p_i//t_o$, $1 \leq i \leq n$, then d is union-free iff there is a query among them, v_j, having an intermediary part p_j such that all other p_i map into p_j.*

Proof: We prove this result by induction on the number of patterns. Lemma 3.A.3 already proves this result for $n = 2$.

As in the case of Lemma 3.A.3, we first rewrite d by rule R1, coalescing the root and result tokens of the parallel branches. At this point, the only rules that remain applicable are R6 and R7.

Let us now assume that some run of Apply-Rules terminates on d without outputting a tree. Then, it is easy to check that Apply-Rules will also stop in the particular run, in which we start by applying only R7 until it does not apply anymore. We show in the following that d resulting from this run is not union-free.

We continue with d obtained, as we said, possibly after some applications of R7 that removed some of the branches in parallel, yielding a DAG pattern as the one illustrated in Figure 3.6. $2 \leq k \leq n$ denotes the number of remaining branches in parallel and $i_1, \ldots i_k$ denote these branches. W.l.g., let these be the intermediary parts of $v_1, \ldots, v_k$ respectively.

Note that we are now in a setting in which $d \equiv v_1 \cap \cdots \cap v_k$ and the following holds:

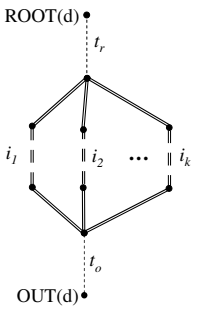


Figure 3.6: DAG pattern d after (possibly) applying R7.

(†) *there is no mapping (not necessarily root-mapping) between the intermediary parts of any* $v_1, \ldots, v_k$.

Next, starting from the DAG pattern d in Figure 3.6, by (†), only rule R6 is applicable. For convenience, we assume that R6 steps are applied by a slightly different strategy: we take an R6 step only if it applies to *all* the parallel branches simultaneously. At some point, this process will stop as well and we obtain a refined d as illustrated in Figure 3.7 (only the main branches are given). The t_i tokens cannot all be equivalent.

Let $p_i = t_i//r_i$ denote the branches in parallel.

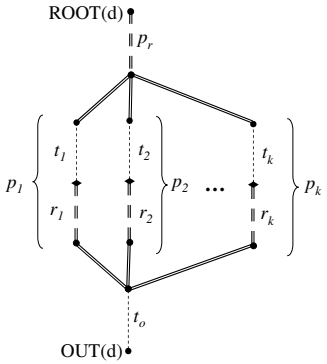


Figure 3.7: DAG pattern d after R6 steps applied to all branches.

Let us assume towards a contradiction that d *is union-free* and let q be the interleaving such that $q \equiv d$. W.l.g., let q be of the form

$$q = p_r//t//m//t_o$$

where t is the token immediately following p_r (the m part might be empty). W.l.g., let us assume that $t_1 \not\equiv t$ (we know that there must be at least be one such token among $t_1, \ldots, t_k$.) We will show that by assuming $q \equiv d$ we arrive at the contradiction that $t \equiv t_1$.

For p_1 chosen in this way, let d' denote the DAG pattern obtained from d by removing its p_1 branch. Introducing for each i the pattern

$$v'_i = p_r//p_i//t_o,$$

by (†) all incomparable, note that d' can be seen as

$$d' = dag(v'_2 \cap \cdots \cap v'_k)$$

and note also that

$$d \equiv d' \cap v'_1 = d' \cap (p_r//t_1//r_1//t_o).$$

By the inductive hypothesis, d' is not union-free, i.e. there are $m \geq 2$ and some patterns q_i, which are some incomparable interleavings of d' (such their root and result tokens have the same main branch), all of the form $p_r// \ldots //t_o$ (by induction, from Claim 3.A.4), such that

$$d' \equiv q_1 \cup \cdots \cup q_m.$$

We can thus conclude that

$$d \equiv v'_1 \cap (q_1 \cup q_2 \cup \cdots \cup q_m) = (v'_1 \cap q_1) \cup (v'_1 \cap q_2) \cup \cdots \cup (v'_1 \cap q_m).$$

Note now that we cannot have $v'_1 \sqsubseteq q_i$, for any q_i, since this would mean that $v'_1 \sqsubseteq v'_2, \ldots, v'_k$, in contradiction with †.

In the following, we will proceed by doing an exhaustive case analysis:

Case 1: *for all q_i, we have $q_i \not\sqsubseteq v'_1$.*

In this case, each intersection of two given above will not be union-free. This follows easily from Lemma 3.A.3 and its Claims since v'_1 and q_i have the same root tokens and result tokens (since there is no containment mapping between them there can be no mapping between their intermediary parts).

Hence any interleaving resulting from some DAG pattern $d_i = dag(q_1 \cap v_i')$ cannot even reduce all the other interleavings of d_i, so d cannot be union-free in this case, since $d = \cup_i d_i$. This case can be thus discarded.

Case 2: *at least two interleavings of d', say q_1 and q_2, are such that $q_1 \sqsubseteq v_1$ and $q_2 \sqsubseteq v_1$.*

We can thus reformulate d as

$$d \equiv q_1 \cup q_2 \cup (v_1' \cap q_3) \cup \cdots \cup (v_1' \cap q_m).$$

Now, each DAG pattern $v_1' \cap q_j$ is not union-free and, moreover, their interleavings cannot contain q_1 or q_2 (since $q_1, q_2 \not\sqsubseteq q_j$ in the first place). Also, obviously, $q_2 \not\sqsubseteq q_1$ and $q_1 \not\sqsubseteq q_2$. So again d can not be union-free and this case can be discarded as well.

Case 3: *exactly one of the interleavings of d' , call it q_1, is contained in v_1' ($q_1 \sqsubseteq v_1'$).*

In this case, d can be reformulated as

$$d \equiv q_1 \cup (v_1' \cap q_2) \cup \cdots \cup (v_1' \cap q_m)$$

and cannot be union-free unless it is in fact equivalent to q_1. This means that for all other q_i's we must have $v_1' \cap q_i \sqsubseteq q_1$. Of course, q_1 should be equivalent (isomorphic modulo minimization, by Lemma 3.A.1) to q, the interleaving of d for which we supposed $d \equiv q$, i.e.

$$q_1 \equiv p_r//t//m//t_o.$$

We continue by assuming for instance that $v_1' \cap q_2 \sqsubseteq q_1$.

Recall that v_1' is of the form

$$v_1' = p_r//p_1//t_o$$

and let q_2 be of the form

$$q_2 = p_r//m_2//t_o.$$

Now, since $q_2 \not\sqsubseteq v_1'$ and they have the same root token and result token, there is no mapping from p_1 into m_2. Consequently, let sf_1 denote the maximal

token-suffix of p_1 that can map into m_2, and let pr_1 denote the remaining part (i.e., a token-prefix). Note that pr_1 cannot be empty. So we can write v'_1 as

$$v'_1 = p_r//pr'_1//sf_1//t_o$$

where pr'_1 is an isomorphic copy of pr_1.

Let us now consider the interleaving u of $v'_1 \cap q_2$, of the form

$$u = p_r//pr''_1//m_2//t_o.$$

where pr''_1 is an isomorphic copy of pr_1 as well.

Since we assumed that $u \sqsubseteq v'_1 \cap q_2 \sqsubseteq q_1$, there must exist a containment mapping ψ from q_1 into u.

Since $q_1 \sqsubseteq v'_1$, let ϕ be a containment mapping from v'_1 into q_1. So we have $v'_1 \xrightarrow{\phi} q_1 \xrightarrow{\psi} u$.

In particular, ϕ must map the $pr'_1//sf_1$ part of v'_1 in the $t//m$ part of q_1. With a slight abuse of notation, let $\phi(pr'_1)$ denote the minimal token-prefix of $t//m$ within which the image under ϕ of pr'_1 occurs. In other words $\phi(pr'_1)$ starts with the first token of t and ends with the token into which the last token of pr'_1 is mapped. (Again, $\phi(pr'_1)$ is well defined because all patterns are skeletons and tokens can only map strictly inside tokens.)

We can thus write q_1 as

$$q_1 = p_r//\phi(pr'_1)\dots\phi(sf_1)//t_o.$$

Next, we argue that in the containment mapping ψ of q_1 into u, we must have $\psi(\phi(pr'_1)) = pr''_1$. (This follows easily from the definition of sf_1.) And this implies that $\phi(pr'_1) \equiv pr''_1 \equiv pr_1$. Hence q_1 and v'_1 start by the some common non-empty token-prefix. Since one of them starts by t and the other by t_1 this means in the end that $t \equiv t_1$. This is a contradiction.

Finally, we can generalize Claim 3.A.4 by the following claim:

Claim 3.A.5. *All the interleavings of nf(d) are of the form*

$$p_r//\dots//t_o.$$

In conclusion, we know for now that APPLY-RULES is complete for the case of DAG patterns that are defined as the intersection of skeleton queries when their root and result tokens have the same main branch. Such an intersection is union-free iff there is a query v_i among them having an intermediary part into which all the other intermediary parts map. If this is not the case, the DAG is equivalent to a union of interleavings having the same root tokens and result tokens, as described in Claim 3.A.5. ■

Lemma 3.A.5. *Given 2 extended skeletons v_1, v_2, where v_1 has only one token, then dag($v_1 \cap v_2$) is union-free iff* APPLY-RULES *rewrites it into a tree.*

Proof: The case in which v_2 has only one token is trivial: we can just apply rule R1 for all edges. Let us assume that v_2 has at least one //-edge in the main branch i.e. it has more than one token.

Suppose that the output of the rewriting algorithm, call it d', is not a tree pattern. It is easy to see that there is a subpattern sd in d' that is not a tree and it has a //-edge, otherwise R1 (plus maybe R7) would have reduced that subgraph to a tree. Then there must be a node n_1 in d' such that there are 2 main branch paths going out of n_1: one starting with a /-edge n_1/n_2 and another one starting with a //-edge $n_1//n_3$, such that the 2 paths meet again starting from a node n_4. We can also infer that on the $n_1//n_3$ branch, the last edge before n_4 is a //-edge $n_5//n_4$, as in Figure 3.8, otherwise R1.ii would have applied or the pattern would have been unsatisfiable.

Since R2 did not apply, it means that n_2 and n_3 are collapsible. Let p_2 be the main branch of the token starting with n_3 in its branch. Let p_1 be the main branch path (with only /-edges) between n_2 and n_4. As n_2 and n_3 are collapsible, it means that either p_2 is a prefix of p_1 or p_1 is a prefix of p_2. Since R3 did not apply, it means that either (i) p_1 is a strict prefix of p_2 or (ii) that p_2 is a prefix of p_1, but there is a predicate Q attached to a node n_7 of p_2 that does not map into p_1. (i) leads to immediate unsatisfiability (a /-path longer than a parallel one). In case (ii), Q must be a predicate that satisfies the definition of extended skeletons but does not satisfy the conditions of R5. Suppose the DAG were union-free. Then there is one interleaving i_1 that contains all others, and, in i_1, n_7 is collapsed into a

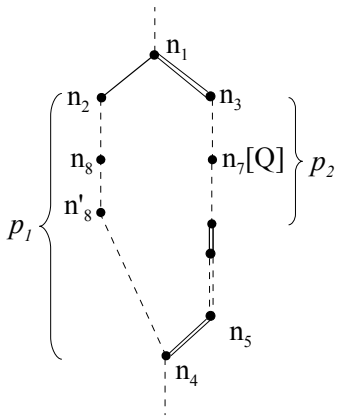


Figure 3.8: Subgraph *sd* (when one query has 1 token).

node n_8 of p_1. If there were only one choice, n_8, for collapsing n_7 into p_1, R8 would have applied. Hence, we have at least one more choice for mapping the branch of n_7. There is only one case that could make Q not satisfy the conditions of R5. This is when some other collapse choice n'_8 of n_7 into p_1, leading to a pattern d', $pattern(\lambda_d(n_8)[Q])$ does not have a root mapping into $\mathrm{SP}_{d'}(n_8)$. This immediately implies that i_1 does not subsume all other collapse choices. Hence the graph cannot be union free. ■

Lemma 3.A.6. *Given n extended skeletons $v_1, v_2, \ldots, v_n$, where v_1 has only one token, $dag(v_1 \cap v_2 \cap \cdots \cap v_n)$ is union-free iff* Apply-Rules *rewrites it into a tree.*

Proof: As for Lemma 3.A.5, w.l.o.g we can consider that $v_2, v_3, \ldots v_n$ have more than one token.

Also, we can infer the existence of a subgraph *sd* with main branches starting in n_1/n_2, $n_1//n_3$ and ending in n_4, with $n_5//n_4$ on the same branch as $n_1//n_3$. An important observation is that, since the n_1/n_2 branch has no //, the DAG can be union free only if the branch with $n_1//n_3$, and in fact any other parallel main branch, maps into the branch of n_1/n_2.

Consider as before the maximal /-path p_2 starting at n_3.

Case A: Suppose first that p_2 forms a tree subpattern in d, but $\mathrm{TP}_d(p_2)$ does not map into $\mathrm{SP}_d(n_2)$ because of a predicate Q on a node n_7 that prevents any

mapping into p_1. And assume the pattern d is union-free, implying the existence of an interleaving i_1 of sd that contains the others. In i_1, n_7 is collapsed with a node n_8 of p_1. Reasoning as for Lemma 3.A.5, since R5 did not apply, Q does not satisfy the condition required by R5: there is a collapse choice n_9 of n_7 into p_1, leading to a pattern d', such that $pattern(\lambda_d(n_8)[Q])$ does not have a root mapping into $\mathrm{SP}_{d'}(n_8)$. Let i_2 be the interleaving in which n_7 merges with n_9. (Please note that all interleavings are /-patterns, because the main branch of one of the queries, v_1, has only /-edges.)

As i_1 (in which Q is on node $n_{7,8}$) maps into i_2 (in which Q is on node $n_{7,9}$), the mapping of Q has to be made possible by some predicate Q' of a node n_{10} from another main branch, neither the one of p_1, nor the one of p_2. n_{10} is merged, in i_2, somewhere below n_8, into a node n_{11}. Thus, Q maps its prefix into i_2 in the path between n_8 and $n_{10,11}$, and then continues to map in Q', as in Figure 3.9 ($n_{7,9}$ can be either above or below $n_{10,11}$).

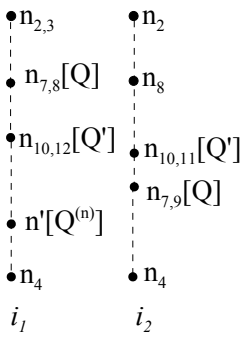


Figure 3.9: Interleavings with a 1-token query.

But Q' does not verify the conditions from R5, otherwise it would have already been copied at n_{10}. Moreover, there is at least another node n_{12} of p_1 in which n_{10} can collapse, otherwise R8 would have applied and Q' would have already been added on n_{10}, making also Q verify the conditions of R5 (contradiction with the previous assumptions). And in order to have an image for Q' in the other interleavings, we will need another predicate Q" and so forth. We can repeat the same argument and show by induction on the number of main branches,

that eventually we find a predicate $Q^{(n)}$ of i_1 that cannot be mapped in all other interleavings, because the number of main branches is finite. Hence there is no interleaving i_1 that makes the pattern union-free.

Case B: Suppose now that p_2 does not form a tree subpattern in d. Let m_1 be the node closest to n_3 in p_2 that intersects another main branch.

Case B.I: Consider the case when there is a downward edge $m_1//m_2$ starting another main branch path. Then the path between n_3 and m_1 is a subtree pattern and we can reduce to *Case A*.

Case B.II: Suppose there is an edge $m_2//m_1$, where m_2 belongs to a different main branch than that of p_2. Then the main branch of $m_2//m_1$ intersects the main branch of n_1/n_2 at the bottom in n_4 and at the top in a node n_1'. Let p_2' be the main branch path following n_1' in the main branch leading to m_2. If p_2' is a subtree pattern, then we can apply Case A to conclude the graph is not union free. If p_2' is a subtree pattern, then either there is another main branch path starting from one of its nodes and going downwards, and then Case B.I would apply, or there is another main branch path that ends in one of its nodes (as for $m_2//m_1$). Since the number of views is finite, we can iterate this process until either Case A or Case B.I applies. ■

***XP* fragment for PTIME.** Now we are ready to give the completeness proof for skeletons.

Proof: (**Theorem 3.3.5**)

We will show that, given n (extended) skeletons $v_1, \ldots, v_n$, Apply-Rules is complete for deciding union-freedom for the DAG pattern $d = dag(v_1 \cap \cdots \cap v_n)$.

We first rewrite d by R1 steps. We obtain after this phase a DAG pattern d in which the root token of d may have several main branch nodes with outgoing //-edges. Similarly, the result token may have several nodes with incoming //-edges. If this is not the case, neither for the root token nor for the result token, then we know that the Apply-Rules is in this case complete by Lemma 3.A.4.

Let us assume that some run of Apply-Rules ends without a tree. We can easily prove that in this case the the following run Apply-Rules would also

stop without yielding a tree:

- first refine by rules R2, R3 and R4 the root token and the result token w.r.t. their outgoing/incoming //- edges,
- then rewrite out some of the branches in parallel by applying R7.

We continue assuming that we do not obtain a tree by the above run. At this point, d is a DAG pattern as the one illustrated in Figure 3.10, where t_r denotes the root token (ending with node n_r) and t_o denotes the result token (starting with n_o). Rules R2, R3 and R4 no longer apply, hence each //-edge outgoing from a node of t_r that is ancestor of n_r cannot be refined into connecting it to a lower node in t_r. The same observation holds for //-edges incoming for nodes of t_o that are descendants of n_o.

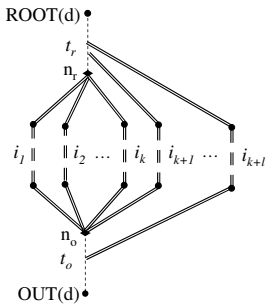


Figure 3.10: DAG pattern d for extended skeletons.

The intermediary branches $i_1, \ldots, i_k$ denote those that start from n_r and end at n_o (we use this notation, even if there may be no such $i_1, \ldots, i_k$ and $k = 0$). The other branches in parallel, $i_{k+1}, \ldots, i_{k+l}$, denote those that do not obey both conditions. If $l = 0$, i.e. there are no such branches, we fall again in the case handled by Lemma 3.A.4, for which the Algorithm is complete. We continue with the assumptions that $k \geq 0$ and $l \geq 1$ as well.

We next prove that d is not union-free.

We introduce some additional notation. For each i_j, $k+1 \leq j \leq k+l$, such that i_j starts above n_r, let n_j^r denote the node in t_r that is sibling of the first node in i_j (i.e., n_j^r and the first node in i_j have the same parent node, a node

in t_r). Note that n_j^r is ancestor-or-self of n_r. Let n_j^o denote the node of t_o that is "parent-sibling" of i_j (they have the same child node). n_j^o is defined if i_j ends below n_o and it is descendant-or-self of n_o.

For each i_j, by pr_j we denote its maximal token-prefix that can map in $\mathrm{TP}_d(n_j^r/\ldots/n_r)$. Similarly, for each i_j by sf_j we denote the maximal token-suffix that can map in $\mathrm{TP}_d(n_o/\ldots/n_j^o)$.

Note that pr_j and sf_j cannot overlap since in this case i_j would have been rewritten away by R7. We can thus write each i_j as

$$i_j = pr_j//m_j//sf_j, \text{ for } k+1 \leq j \leq l+1.$$

Now, we consider a second DAG pattern d' obtained from d by replacing each i_j branch by the branch m_j, connected this time by //-edges to n_r and n_o (see Figure 3.11), instead of the parent of n_j^r and the child of n_j^o.

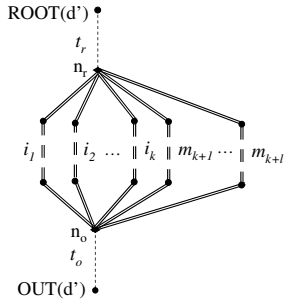


Figure 3.11: DAG pattern d' (for extended skeletons).

We can prove the following:

Claim 3.A.6. *The set of interleavings of d' is included in the set of interleavings of d (set inclusion) and d is union-free only if d' is union-free. If $d' \equiv p$, for an interleaving p, then p is the only candidate for $d \equiv p$.*

Proof of Claim 3.A.6: It is straightforward that all the interleavings of d' are interleavings of d as well. The particularity of d' is that its interleavings do not modify the tokens t_r and t_o. More precisely each interleaving will be of the form $t_r//\ldots//t_o$. Moreover, by the way d' was defined and given that no R2, R3 or R4 steps applied on d, we argue that all other interleavings of d will either (a)

be redundant, i.e. contained in those of d', (b) add some predicate on t_r or t_o or (c) have a longer root token (resp. result token) than t_r (resp. t_o).

But this means that an interleaving $p \in nf(d) - nf(d')$ cannot have a containment mapping into an interleaving of the form $t_r//\ldots//t_o$. Hence it cannot be equivalent to d. So the only interleaving p candidates for $p \equiv d$ are those of $nf(d')$. From this it follows that d can be union-free only if d' is union-free.

We continue the proof of Theorem 3.3.5. Note that by Lemma 3.A.4 d' is union-free iff there exists some m_j into which all $i_1, \ldots, i_k$ and all other m_i's map. This is because of the assumption that among $i_1, \ldots, i_k$ there is no branch i_j into which all other i_i's map.

We continue towards showing that d is not union-free with this assumption and let m denote the branch into which all others map. Note that among $m_{k+1}, \ldots, m_{k+l}$ there can be more than one "copy" of m (i.e., equivalent to m). By m_c we denote all these copies. Among $i_1, \ldots, i_k$ there is no copy of m (otherwise R7 would have triggered).

Let $d' \equiv p = t_r//m//t_o,\ for\ m = t//m'$. We build next an interleaving w of d such that $w \not\sqsubseteq p$.

W.l.g. let us assume that all the m_c copies of m are connected in d to a node that is strict ancestor of n_r[2]. Since R2 or R4 did not apply on these copies of m, it means that a strict prefix of the main branch of m's first token t maps in a suffix of the main branch of t_r, when the possibly non-empty preceding token-prefix pr_j is collapsed somewhere "higher".

Let ψ denote the partial mapping from t into t_r that uses the maximal possible prefix of t across all the copies m_c. Let t be $t = t'/t''$, where t' is this maximal prefix (not empty).

[2]The remaining cases when

- all the m_c copies of m are connected in d to a node that is strict descendant of n_o, or
- all the m_c copies of m but one (we cannot have more than one, otherwise R7 would have triggered leaving only one) are connected in d to a node that is strict ancestor of n_r and all the m_c copies of m but one are connected in d to a node that is strict descendant of n_o,

can be handled similarly.

We are now ready to build w.

We build first the root token t'_r of the w interleaving as follows: let t'_r denote an interleaving of t_r and t defined by the code $i = \mathrm{MB}(t_r)/\mathrm{MB}(t'')$, and f_i defined as "identity" on t_r and t'', and $f_i(n) = \psi(n)$ for the main branch nodes of t'.

We build the intermediary part p of the w interleaving as follows: starting from $i_x \in \{i_1, \ldots, i_k, m'\}$, (or simply from $i_x \in \{i_1 \ldots, i_k\}$ in the case m' is empty), , let us interpret them as the intermediary parts of the following skeleton patterns

$$s_x = start//i_x//end.$$

Let also s denote the pattern

$$s = start//m//end.$$

Let us now consider now the DAG pattern $d' = dag(\cap_x s_x)$. Since none of the s_x patterns is equivalent to s, from Lemma 3.A.4 we have that $d' \not\equiv s$. Moreover, since $s \sqsubseteq d'$ (because $s_x \sqsubseteq s$), we must have that $d' \not\sqsubseteq s$. In other words, there must exist an interleaving w' of d', of the form $start//p//end$ such that $w' \not\sqsubseteq s$. Finally, this means p is such that while all the $i_1, \ldots i_k, m'$ map into it, we have that m does not map into it.

Finally, we define w:

$$u = t'_r//p//t_o$$

It is easy to check that u is an interleaving of d (d has a containment mapping into u) but $u \not\sqsubseteq p = t_r//t//m'//t_o$. Hence d is not union-free.

In order to finish the proof, we also need to handle the case in which one of the skeleton queries has only one token. This is done by Lemma 3.A.6. ■

Let us summarize the conclusions of the proof of Theorem 3.3.5: After R1 steps, followed eventually by R2, R3 and R4 steps, we obtain the branches in parallel $i_1, \ldots, i_k$ starting from the last node of the root token (t_r) and ending with the first node of the result token (t_o). Other branches in parallel may exist in d, but connected to other nodes of t_r and t_o. Then, by eventually some R7 steps, the DAG pattern must become a tree, otherwise it is not union-free. The resulting tree is $t_r//i_1//t_o$, where i_1 is one of the branches in parallel (into which all other, $i_2, \ldots, i_k$ map).

We go now beyond extended skeletons, taking into account predicates that start by a //-edge. We adopt the following approach: assuming that the DAG rewriting process stops outputting a DAG pattern d that is not a tree, we identify a set of candidate interleavings $c \in \mathit{interleave}(d)$ such that d is union-free only if one of them is equivalent to d. Then, for each candidate c we build an interleaving $w \in \mathit{interleave}(d)$ that is not contained in c. This is sufficient to conclude that d cannot be union-free, hence the Algorithm is also complete.

Rewrite-plans for PTIME. Now we are ready to give the completeness proofs for akin patterns.

Proof: (**Theorem 3.3.6**)

We will show that, given n akin tree patterns $v_1, \ldots, v_n$, Apply-Rules decides union-freedom for $d = dag(v_1 \cap \cdots \cap v_n)$.

Let each v_j be defined as $v_j = t_r^j//i_j//t_o^j$.

Special case. We start by considering the special case when the patterns have the same main branch for their result tokens as well.

By Lemmas 3.A.2 and 3.A.4, we know that d is union-free only if the intermediary parts i_j are such that their skeletons map in the skeleton of one of them. Without loss of generality, let us assume that all $s(i_j)$ map in $s(i_1)$. continue with this assumption.

First, the initial R1 steps coalesce the root and result tokens of the n branches, yielding a DAG pattern similar to the one illustrated in Figure 3.6. Then, the only rules that may be applicable are R6 and R7. Let us assume that Apply-Rules stops outputting a pattern that is not a tree. We show that d is not union-free.

If the algorithm Apply-Rules stops without outputting a tree in some run, then it will also stop without outputting a tree in the following particular rewriting strategy

- we first apply R6 on the "biggest"branches in parallel i_j, i_k such that $s(i_j) \equiv s(i_k) \equiv s(i_1)$. It is straightforward that R6 must apply for these branches, coalescing entirely the two branches into one branch. After this phase, there

will be no other parallel branch with skeleton $s(i_1)$, besides i_1 itself.

- Then, R6 is applied only on all the branches in parallel at once. This phase will terminate with a refined d similar to the one illustrated in Figure 3.7, where $2 \leq k \leq n$, p_r denotes the common part following the root (may have several tokens if R6 was applied) and $t_1, \ldots t_k$ denote the sibling tokens on which R6 no longer applies (i.e., they are not all *similar* hence they do not all have the same skeleton).

  Also, rule R7 is applied freely, and it can rewrite out some of the branches in parallel.

  After this phase, while there exists a mapping from each $s(p_i)$ into $s(p_1)$, there is no mapping from p_i into p_1 . Note also that, by the first phase of the rewriting strategy, we cannot have the opposite mapping from $s(p_1)$ into $s(p_i)$.

- Finally, rule R6 is applied only between p_1 on the one hand, and other branches p_i on the other hand, while R7 is still applied freely.

We obtain in the end a DAG pattern similar to the one illustrated in Figure 3.12. Let us assume that besides p_1 there are l remaining branches in parallel, connected by a //-edge either to p_r or to various tokens of p_1.

Let $p_1 = t_1//\ldots//t_m$. For each $i = \overline{1,l}$, let $p'_i = t'_i//r'_i$ denote now these branches in parallel with (part of) p_1. For each $i = 1, l$, let t_i^1 denote the token in p_1 that is sibling of the token t'_i. Note that t'_i and t_i^1 must be dissimilar, hence will have different skeletons. Let sf_i^1 denote the token-suffix of p_1 that is in parallel with p'_i and let pr_i^1 denote the rest of p_1 (a token-prefix). For each $i = 1, l$, we can thus reformulate p_1 as $p_1 = pr_i^1//sf_i^1$, where the first token of sf_i^1 is t_i^1. Note that for each i we have that $s(p'_i)$ maps into $s(sf_i^1)$, while the opposite is not true.

Next, we can prove the following claim:

Claim 3.A.7. *d can be union-free, for some c such that $d \equiv c$, only if c is of the form $c = p_r//m//t_o$ where $s(m) = s(p_1)$.*

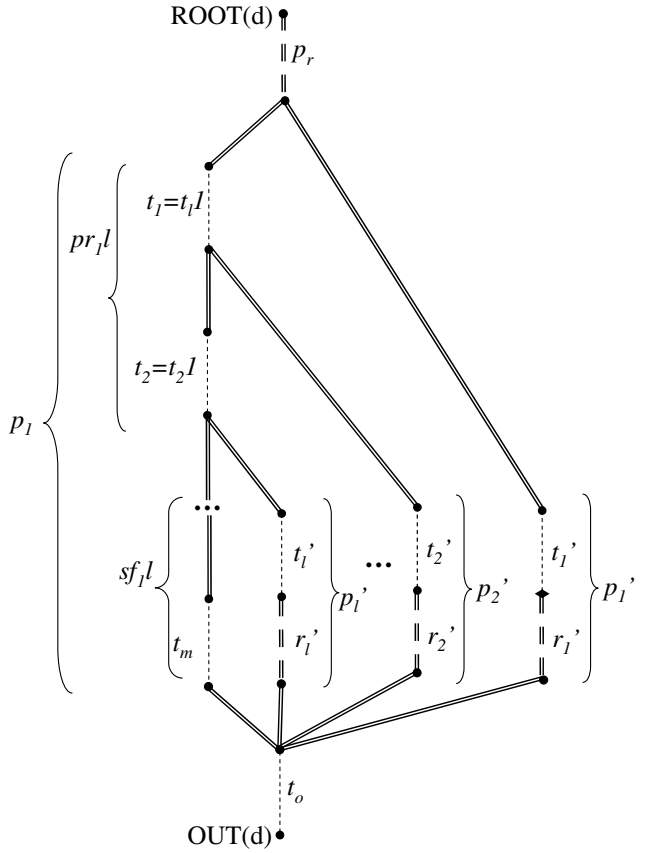


Figure 3.12: DAG pattern d' (for rewrite plans).

Proof of the claim. This is the minimal skeleton for an interleaving.

All the candidate interleavings c will be defined by the code $i = \text{MB}(p_r//m//t_o)$ and some function $f_i : \text{MBN}(d) \to i$. What distinguishes the various c's is the definition of the f_i function on the nodes of the branches, $p'_1, \ldots, p'_l$ (since the other main branch nodes in d have only one possible image). We will show that for any such f_i and associated interleaving c we can build the w witness such that $w \not\sqsubseteq c$.

Let f_i be fixed and let c denote the corresponding interleaving for code i and function f_i. Note that we can interpret f_i as a series of rewrite steps over d that collapse the pairs of nodes $(n, f_i(n))$, for all the nodes n in the $p'_1, \ldots, p'_l$ branches, outputting as end result the tree pattern c. These steps do not modify

the skeleton of p_1, hence can only bring some new predicates starting by //-edge.

Next, we describe how the interleaving $w \not\sqsubseteq c$ is built, from the current DAG pattern d of Figure 3.12.

Let $n_c \in \text{MB}(c)$ denote the lowest main branch node in c's m part which has a subtree predicate st that is not present (in other words, cannot be mapped) at the associated node n_1 in the p_1 part of d. st must start with a //-edge and must come from a node of some (maybe several) branches p'_i. (We know that such a node n_c must exist, otherwise the p'_i branches would fully map in the corresponding branch in parallel sf_i^1 and rule R7 would have applied).

Without loss of generality, let $n'_{i_1}, \ldots, n'_{i_s}$, for $\{i_1, \ldots, i_s\} \subseteq \{1, \ldots, l\}$, denote the nodes from the branches $p'_{i_1}, \ldots, p'_{i_s}$ that are the "source" of st[3]. So we have $f_i(n_1) = f_i(n'_{i_1}) = \cdots = f_i(n'_{i_s}) = n_c$ and we can say that n_c is the result of coalescing n_1 with $n'_{i_1}, \ldots n'_{i_s}$.

Now, we can see the left branch p_1 as being divided into two parts, the one down to the token of n_1 (that token included), denoted p_{11}, and the rest, denoted p_{12}. So we can write p_1 as

$$p_1 = p_{11}//p_{12}.$$

Similarly, for each $x \in \{i_1, \ldots, i_s\}$ we can see each main branch $pr_x^1//p'_x$ as being divided into two parts, the one down to the token of n'_x (that token included), denoted p'_{x1}, and the rest, denoted p'_{x2}. So we can write each main branch $pr_x^1//p'_x$ of d as

$$pr_x^1//p'_x = p'_{x1}//p'_{x2}.$$

Note that by the way n_c was chosen (as the lowest node) we can conclude that by f_i (on the main branch nodes) we can fully map $\text{TP}_d(p'_{x2})$ into $\text{SP}_d(n_1)$, for all x (i.e., there are no other added predicates below n_1's level). It is also easy to see that while $s(p'_{x1})$ maps in $s(p_{11})$ (by f_i), the opposite is not true, otherwise R6 steps would have applied up to this point.

We are now ready to construct w. First, we obtain a part p of the w interleaving as follows: starting from the set of skeleton queries $s(p'_{x1})$, for all $x \in \{i_1, \ldots, i_s\}$, let us interpret them as the intermediary parts of the following

[3]They have a predicate st' into which st maps.

skeleton patterns

$$s_x = start//s(p'_{x1})//end.$$

Let also s denote the skeleton pattern

$$s = start//s(p_{11})//end$$

Let us now consider the DAG pattern $d' = dag(\cap_x s_x)$. Since none of the skeleton patterns s_x is equivalent to s, from Lemma 3.A.4 we have that $d' \not\equiv s$. Moreover, since $s \sqsubseteq d'$ (because $s \sqsubseteq s_x$, by the way c was defined), we must have that $d' \not\sqsubseteq s$. In other words, there must exist an interleaving w' of d', of the form $start//p//end$ such that $w' \not\sqsubseteq s$. Finally, this means p is such that while all the $s(p'_{x1})$ map into it, we have that $s(p_{11})$ does not map into it. We will use this property. For each p'_{x1}, let f_{x1} denote a mapping from $s(p'_{x1})$ into p.

Next, we obtain a second part of w as follows. Let pr_{11} denote the maximal token-suffix of p_{11} such that $s(p_{11})$ can map in p, and let sf_{11} denote the remaining part. sf_{11} cannot be empty, so it is formed by at least the last token of p_{11}, the one with node n_1. So we can see p_{11} as

$$p_{11} = pr_{11}//sf_{11}.$$

Let f_p denote a partial mapping from $s(p_{11})$ into p that exhibits sf_{11}.

We will define w by a code i' and function f'_i as follows:

- $i' = \lambda(p_r//p//sf_{11}//p_{12}//t_o)$,
- f'_i maps nodes of MBN(d) into i' positions as follows:
  - f'_i is "identity" for the main branch nodes of p_r, t_o, for the sf_{11} part of the p_{11} prefix of p_1 and for the p_{12} suffix of p_1,
  - for the remaining main branch nodes n in p_{11} (i.e., those of pr_{11}), $f'_i(n) = f_p(n)$,
  - for the main branch nodes n of the p'_{x1} prefix of the $pr^1_x//p'_x$ branch in d, for $x \in \{i_1, \ldots, i_s\}$, $f'_i(n) = f_{x1}(n)$

- for the remaining nodes n in the $pr_x^1//p'_x$ branches (i.e. those in p'_{x2}), $f'_i(n) = f'_i(f_i(n))$.
- finally, for all the main branch nodes of the remaining branches p'_y, for $y \notin \{i_1, \ldots, i_s\}$, $f'_i(n) = f'_i(f_i(n))$.
  (they go where their images under f_i go.)

Claim 3.A.8. *w is an interleaving of d and $w \not\sqsubseteq c$.*

Proof of the claim. First, it is easy to check that w is an interleaving for d. Recall that c is such that

$$s(c) = s(p_r//p_1//t_o) = s(p_r//pr_{11}//sf_{11}//p_{12}//t_o).$$

Second, it is also easy to check that c can have a containment mapping in w iff its sf_{11} part maps in the sf_{11} of w. But this is not possible because the *st* subtree predicate is not present on the $f'_i(n_1)$ node of w (which is found somewhere in the last token of the sf_{11} part).

General case. We now consider the general case, when the result tokens do not necessarily have the same main branch. After the possible rewrite R1(i) steps on the root tokens, and after the possible rewrite steps of R1(ii), R2(ii), R3(ii) and R4(ii) on the result tokens, we may now obtain a DAG pattern in which the branches in parallel may not be "connected" to t_o at its highest node (n_o), but at some other node that is strict descendant of n_o. If this is not the case, then we are back to the special case discussed previously.

Otherwise, let us now consider the DAG pattern d' obtained from d by connecting the endpoints of the branches in parallel at n_o. We can easily see that the interleavings of d' are all among those of d and moreover, d is union-free only if d' is union-free, with $d' \equiv d \equiv p$, for some $p \in interleave(d')$. This is because the interleavings of d that are not interleavings of d' as well are those that add some predicates on t_o that are not present in all the interleavings.

By Lemmas 3.A.2 and 3.A.4, we know that d is union-free only if the intermediary parts i_j are such that their skeletons map in the skeleton of one of

them, which in addition, in the current d pattern, must start at n_r and end at n_o. Without loss of generality, let us assume that this is i_1 (note that the i_1 branch will not be affected by the transformation from d to d').

From the special case, we know under what conditions d' is union-free, and it is immediate that, when they hold, the interleaving p is obtained from i_1 possibly by adding some predicates of the form $[//\ldots]$ to some of its main branch nodes. Importantly, each such predicate is added on the highest possible main branch node of i_1.

Finally, it is now easy to check that an interleaving p of d' obtained in this way will always have a containment mapping in any interleaving p' of d: everything except the added predicates will map (by identity), while the added predicates (of the form $[//\ldots]$) will map in their respective occurrence in p' (by necessity, found at a lower main branch node then in the one in p).

This ends the completeness proof for akin patterns in $XP_{//}$. ■

3.B Beyond Tractability

We give first the proofs for Theorem 3.3.7 (1) and (2). Then we show how we construct similar reductions to prove Theorem 3.3.8 (1) and (2).

Proof: (**Theorem 3.3.7 (1)**) To show that our problem is in coNP we use the following approach: we show that one can always build an interleaving p_c that is the *unique* candidate for $p_c \equiv d$. Then, we can use an argument similar to the one used in the proof of Theorem 3.3.3, to check in coNP if $d \sqsubseteq p_c$.

We start with d being a DAG pattern as the one illustrated in Figure 3.10 (after R1 steps on the root and result tokens). We have some branches in parallel $i_1, \ldots, i_k, i_{k+1}, \ldots, i_{k+l}$, starting and ending at various nodes of t_r and t_o. We proceed towards building p_c.

If $l = 0$ (i.e. all the branches start from the last node of t_r and end at first node of t_o) we can jump directly to after the claim, with $d' = d$.

If not, we will do first some manipulations on d. We use the same notation as in the proof of Theorem 3.3.5. For each i_j, $j = k+1, k+l$, n_j^r denotes the node

in t_r that is sibling of the first node in i_j, and n_j^o denotes the node of t_o that is "parent-sibling" of i_j (they have the same child node).

Now, for each i_j, $j = k+1, k+l$, by pr_j we denote its maximal token-prefix such that *its extended skeleton* $s(pr_j)$ maps in $\mathrm{TP}_d(n_j^r/\ldots/n_r)$. Note that some of the predicates, among those starting by a //-edge, may no map so we may not have a full mapping from $s(pr_j)$ as well.

Then, for each i_j by sf_j we denote the maximal token-suffix such that sf_j *fully* maps in $\mathrm{TP}_d(n_o/\ldots/n_j^o)$.

We can thus write each i_j as

$$i_j = pr_j//m_j//sf_j,\ for\ j = k+1, l+1.$$

If, for some i_j, pr_j and sf_j overlap then in this case the m_j part is considered empty.

Now, we consider a second DAG pattern d' obtained from d by replacing each i_j branch by the branch m_j, connected this time by //-edges to n_r and n_o (similar to Figure 3.11), instead of the parent of n_j^r and the child of n_j^o.

We can prove the following:

Claim 3.B.1. *d is union-free only if d' is union-free. If $d' \equiv p_c$, then p_c is the only interleaving candidate for $p_c \equiv d$.*

Proof of the claim: This is because all other interleavings are either subsumed by those of d', or change the extended skeleton of t_r, or add some predicates on t_o that will not be present in all interleavings (the candidate must end exactly by $//t_o$).

To simplify presentation let us in the following rename $m_{k+1}, \ldots, m_{k+l}$ as $i_{k+1}, \ldots, i_{k+l}$ and then let $m = k + l$. So we now have the branches in parallel $i_1, \ldots, i_m$.

Next, we know that d' can be union free only if there exists some branch, w.l.g. i_1, such that $s(i_j)$ maps into $s(i_1)$ for all $j = 1, m$. Moreover, it is now easy to see that p_c can only have an extended skeleton of the form $s(t_r)//s(i_1)//s(t_o)$

(this is the minimal skeleton). So, all the branches map their extended skeletons into the branch i_1, while predicates starting by a //-edge may not map.

Note that several choices for mapping each $s(i_j)$ into i_1 may be available, and interleavings that do not use one of these choices cannot lead to p_c (again, they will modify the candidate extended skeleton).

Among those that do not modify the candidate extended skeleton, it now easy to obtain p_c:

For each i_j let ψ_j denote the mapping of $s(i_j)$ in i_1 that uses the *highest possible image* for each token in $s(i_j)$. We build p_c from d' by transformation steps that coalesce main branch nodes as follows: for each $n \in MB(i_j)$, we coalesce the node n with the node $\psi_j(n)$.

We argue that the tree pattern p_c obtained in this way is the only interleaving candidate for $p_c \equiv d'$ (and for $p_c \equiv d$).

Claim 3.B.2. *d is union-free only if $p_c \equiv d$.*

Proof of the claim: This is because all other interleavings that do not modify the candidate extended skeleton will be subsumed (as contained interleavings) by this one. This is because all the predicates that were missing from i_1 and were added by coalesce steps must start with a //-edge. If they would be put below the first possible image of their main branch node, they would anyway be inherited to the main branch nodes above.

The proof that union-free is coNP-easy for $XP_{//}$ is almost done. Now, since we have a clear candidate p_c, obtained in polynomial time, we can guess a witness interleaving p_w of d such that $p_w \not\sqsubseteq p_c$ in polynomial time.

coNP-hardness. The hardness part is proven by reduction from tautology of 3DNF formulas, which is known to be coNP-complete. We start from a 3DNF formula $\phi(\bar{x}) = C_1(\bar{x}) \vee C_2(\bar{x}) \vee \ldots C_m(\bar{x})$ over the boolean variables $\bar{x} = (x_1, \ldots x_n)$, where $C_i(\bar{x})$ are conjunctions of literals.

Out of ϕ, we build patterns $p_0, p_1, \ldots, p_n \in XP_{//}$ over

$$\Sigma = \{x_1, \ldots, x_n, a, c, yes, out\}$$

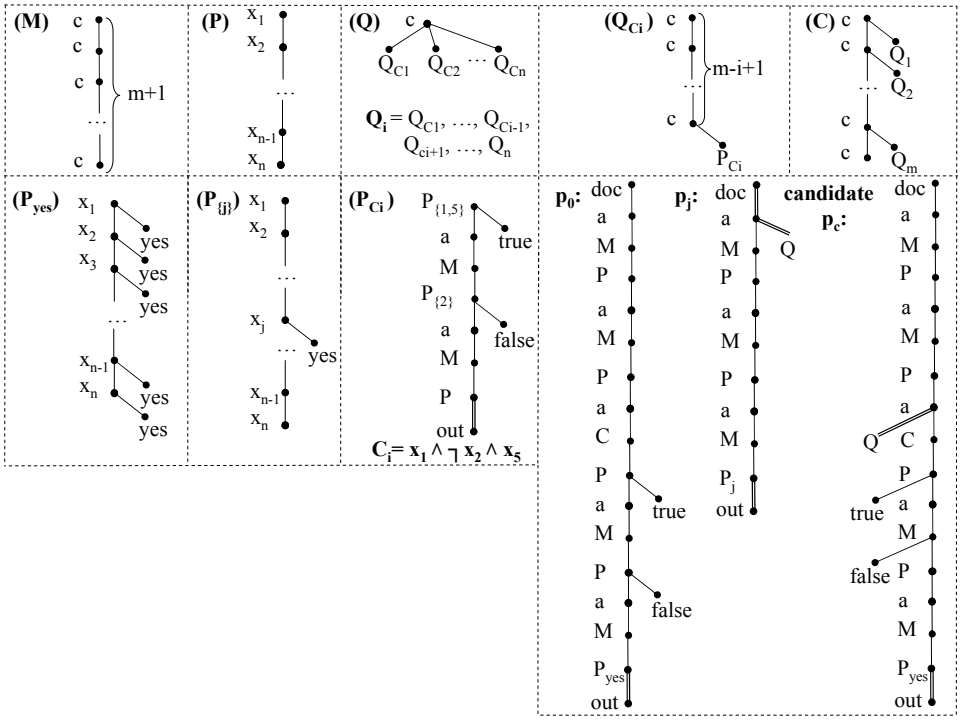


Figure 3.13: The construction for coNP-hardness of union-freedom ($XP_{//}$).

such that the DAG pattern $d = p_0 \cap p_1 \cap \cdots \cap p_n$ is union-free iff ϕ is a tautology.

We build the patterns p_0, $p_1, \ldots, p_n$, based on the gadgets

$$P, P_{yes}, P_{C_1}, \ldots P_{C_m}, M, Q_{C_1}, \ldots, Q_{C_m}, C, Q,$$

and P_X where X can be any set of one, two or three variables (see Figure 3.13).

More precisely, these gadgets are defined as follows:

1. the linear pattern with $m+1$ c-nodes, $c/c/\ldots/c$ (denoted M)
2. the pattern $x_1/x_2/\ldots/x_n$ (denoted P)
3. the pattern $x_1[yes]/x_2[yes]/\ldots/x_n[yes]$ (denoted P_{yes})
4. patterns P_X, where X is a set of variables of size at most 3, obtained from P by putting a $[yes]$ predicate below the nodes labeled by the variables in X.
5. for each clause C_i, the pattern $P_{X_t}[true]/a/M/P_{X_f}[false]/a/M/P//out$, where X_t is the set of positive variables in C_i and X_f is the set of negated variables

in C_i (this pattern is denoted P_{C_i}). For instance, for $C_i = (x_1 \wedge \bar{x_2} \wedge x_5)$, we have the pattern $P_{C_i} = P_{\{1,5\}}[true]/a/M/P_{\{2\}}[false]/a/M/P//out$.

6. for each clause C_i, Q_{C_i} denotes the predicate $[c/c/c/\ldots/c[P_{C_i}]]$, with $m-i+1$ c-nodes,

7. for each C_i, the predicate $Q_i = [Q_{C_1}, \ldots Q_{C_{i-1}}, Q_{C_{i+1}}, \ldots, Q_{C_m}]$, that is the list of all Q_{C_j} predicates for $j \neq i$.

8. the pattern $c[Q_1]/c[Q_2]/c[Q_3]/\ldots c[Q_m]/c$ (denoted C)

9. the predicate $Q = [Q_{C_1}, \ldots, Q_{C_m}]$

The $n+1$ patterns are then given the last section of Figure 3.13.

First, note that no inheritance of predicates occurs in these patters. Q_{C_i} predicates are not inherited in the C part of p_0 because that would require some x_1-label to be equated with the c-label. Similarly, the P_{yes} part of the main branch does not put implicit Q_{C_i} predicates at c-nodes either.

We argue that the candidate interleaving p_c such that $p_c \equiv d$ is unique: p_c is obtained by the code i corresponding to the main branch of p_0, and the function f_i that maps the first a-node (the one with a predicate $[.//Q]$) of each pattern $p_1, \ldots, p_n$ in the same image as the third a-node of p_0 (the parent of the C part). This is the interleaving that will yield the "minimal" extended skeleton (namely the one of p_0), since nodes with a $[yes]$ predicate are coalesced with p_0 nodes having already that predicate. All others would at least have additional $[yes]$ predicate branches and even longer main branches and thus cannot not map into p_c. Hence no other interleaving can contain p_c.

We show in the following that p_c will contain (and reduce) all other interleavings of $p_0 \cap \cdots \cap p_n$ iff ϕ is a tautology. Moreover, it is easy see that p_c contains some interleaving p if and only if its $[.//Q]$ predicate can be mapped at the third a-node from the root in p.

Note now that p_c will contain all interleavings p that for at least some pattern p_j "put" its first a-node either below or in the third a-node of p_0. This

is because $[.//Q]$ would be either explicitly present at the third a-node in p or it would be inherited by this node from some a-labeled descendant.

So, the interleavings that remain be considered are those described by a function f_i' which takes *all* the first a-nodes from $p_1, \ldots, p_n$ higher in p_0, i.e. in either the first or the second a-node of p_0. Each of these interleaving will basically make a choice between these two a-nodes.

For some p_j, by choosing to coalesce its first a-node with the first a-node of p_0 we get an $[yes]$ predicate at the x_i node of the *true* P part of p_0. Similarly, by coalescing with the second a-node we get an $[yes]$ predicate at the x_i node of the *false* P part of p_0. So, these n individual choices of where to coalesce a-nodes amount to a truth assignment for the n variables, and in each interleaving the yes predicate will indicate that assignment.

Recall that in order for p_c to contain such an interleaving p, it must be possible to map the predicate $[.//Q]$ of the third a-node of p_c at the third a-node of p.

We can now argue that $p_0 \cap p_1 \cap \cdots \cap p_n \sqsubseteq p_c$ iff ϕ is a tautology. The if direction (when each truth assignment t makes at least one clause C_i true) is immediate. For a truth assignment with clause C_i being true, in the corresponding interleaving p, the P_{C_i} predicate will hold at the last c-node in the C part, hence the Q_{C_i} predicate will hold at the ith c-node in C. Since all other Q_{C_j} predicates, for $j \neq i$, were already explicitly present at this ith c-node, it is now easy to see that the $[.//Q]$ predicate would be verified at the a-labeled ancestor. Hence there exists a containment mapping from p_c into p.

The only if direction is similar. If for some truth assignment, none of the clauses is TRUE (in the case ϕ is not a tautology), then it is easy to check that p_c will not have a containment mapping into the interleaving p corresponding to that truth assignment. This is because the $[.//Q]$ predicate would not map at the third a-node in p. This concludes the proof. ■

Proof: (**Theorem 3.3.7 (2)**) The proof is similar to the one of **Theorem 3.3.7 (1)**. ■

Proof: (**Theorem 3.3.8 (1)**) First, let us prove the upper bound. For each node of MB(q) in which some of the views $\mathcal{V}_1 \subseteq \mathcal{V}$ map, it is enough to guess one interleaving of $\bigcap_{\mathcal{V}_1} v_j$ in which q does not map. If we put together a polynomial number of polynomially large witnesses, they make up a polynomial witness for the entire problem. In other words, one can verify in polynomial time that there is no rewriting.

For coNP-hardness, we will use the same construction as in the proof of Theorem 3.3.7(1), for a reduction from tautology of a formula ϕ. We define $n+1$ views, $v_0 = p_0$, $v_1 = p_1, \ldots, v_n = p_n$. We define q as $q = p_c$, for p_c being the unique candidate interleaving for the DAG $d = dag(unfold(v_0 \cap v_1 \cap \cdots \cap v_n))$. Moreover, it is easy to see that the only rewrite plan that has chances to be a rewriting is $r = v_0 \cap v_1 \cap \cdots \cap v_n$ (the output node of each view can only be mapped in the output node of q). From this, it follows that r is an equivalent rewriting iff $d \equiv q$ iff d is union-free iff the formula ϕ is valid. This shows that deciding the existence of an equivalent rewriting r for queries and view from $XP_{//}$is coNP-hard. ■

Proof: (**Theorem 3.3.8 (2)**) The fact that the problem is in coNP was already discussed (see proof of Theorem 3.3.8(1)).

For coNP-hardness, we will use the same construction as in the proof of Theorem 3.3.7(2), for a reduction from tautology of a formula ϕ. ■

Chapter 3, in part, is a reprint of the material as it appears in WebDB 2008. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 4

Compactly Encoded Relational Views

ABSTRACT OF THE CHAPTER

In this chapter, I revisit a classical setting in which the user queries are conjunctive queries and the source publishes families of conjunctive queries specified as the expansions of a (potentially recursive) Datalog program with parameters. I present the first study of expressibility and support for sources that satisfy integrity constraints, which is generally the case in practice.

I identify practically relevant restrictions on the program specifying the views that ensure decidability under a mix of key and weakly acyclic foreign key constraints, and beyond. I show that these restrictions are as permissive as possible, since their slightest relaxation leads to undecidability. I present an algorithm that is guaranteed to be sound when applied to unrestricted input and in addition complete under the restrictions.

As a side-effect of this investigation, I settle two problems left open by prior work even while ignoring constraints. First, I improve the previously best known upper bound for deciding support in the constraint-free case, characterizing for the first time the problem's complexity. Second, I establish the precise relationship between expressibility and support: I show them to be inter-reducible in PTIME in both the absence and the presence of constraints.

4.1 Introduction

In this chapter, we revisit the setting of [LRU99, VP00], in which the application queries are conjunctive queries and the source accepts families of possibly parameterized conjunctive queries specified as the expansions of a (potentially recursive) Datalog program. The program is said to *generate* these services, which we will also called *views*. As argued in [LRU99, VP00] and illustrated below, the choice of Datalog as the service specification formalism enables concise yet expressive descriptions of large (even infinite) sets of services over a given schema.

As in Chapter 1, we say that query Q is *expressible* by the program $\mathcal{P}$ if it is equivalent to some view generated by $\mathcal{P}$. Expressible queries can therefore be evaluated at the source, requiring no post-processing at the adapter or client. Q is *supported* by $\mathcal{P}$ if it has an equivalent rewriting R using some finite set $\mathcal{V}$ of views generated by $\mathcal{P}$. Note that finding such R and $\mathcal{V}$ witnessing support enables the following execution plan at the adapter: call the Web services implementing the queries in $\mathcal{V}$, materialize their results locally and run query R over the materialized database.

The challenge in deciding expressibility and support lies in the fact that the family of views to pick from can be very large or even infinite. This renders infeasible any systematic enumeration of views. Remarkably, the two problems were previously shown to be decidable [LRU99], however only when ignoring any knowledge of constraints satisfied by the source. In this work, we investigate the effect of source constraints.

The following example shows that source constraints generate new opportunities for detecting support, calling for algorithms which exploit them. (Example 4.1.1 illustrates a limited-query-capability setting and will be our running example in this chapter.)

EXAMPLE 4.1.1. *Consider a travel information source conforming to the following schema:*

flight(origin, destination) *shuttle(origin, destination)*
train(origin, destination) *bus(origin, destination).*

The source admits only views concerning arbitrary-length itineraries by plane, such that Paris is reachable by train or bus from the destination airport. This family of views is described as the set of all expansions of the distinguished IDB predicate ans *in program* $\mathcal{P}$ *below:*

$$
\begin{aligned}
ans(A, B) &:- f(A, C), ind(C, B) \\
ind(C, B) &:- f(C, B), b(B, \text{“}Paris\text{”}) \\
ind(C, B) &:- f(C, C'), ind(C', B) \\
ind(C, B) &:- f(C, B), t(B, \text{“}Paris\text{”})
\end{aligned}
$$

This example is essentially the same as the example from Section 1.2.3, but written more compactly, for presentation purposes.

Consider a query that asks for 2-leg itineraries ending in an airport from which Paris is reachable by train, bus and shuttle.

$$
\begin{aligned}
Q: \quad q(A, B) :- \; & f(A, C), f(C, B), t(B, \text{“}Paris\text{”}), \\
& b(B, \text{“}Paris\text{”}), s(B, \text{“}Paris\text{”})
\end{aligned}
$$

Clearly, Q *is neither expressible nor supported by* $\mathcal{P}$ *because the views generated by* $\mathcal{P}$ *do not even mention shuttle information. However, suppose we knew the following constraint to hold on the source (stating that any city pair connected by train and bus is also connected by shuttle):*

$$\forall A, S \;\; t(A, S) \wedge b(A, S) \longrightarrow s(A, S). \tag{4.1}$$

Then we would like the wrapper to find the rewriting

$$(R) \qquad r(A, B) :- V_1^b(A, B), V_1^t(A, B)$$

where $\{V_i^b\}_{i\geq 1}$ *(resp.* $\{V_i^t\}_{i\geq 1}$*) are families of views generated by* $\mathcal{P}$*, returning endpoints of itineraries of* i *flight legs where the destination has a bus link (resp. a train link) to Paris. Indeed, it can be checked that* R *is equivalent to* Q *on all databases satisfying (4.1). Therefore* Q *is supported by* $\mathcal{P}$ *when (4.1) holds.*

The problem of deciding support is also of interest for implementing security policies. For security reasons, a source would only allow data access via a set of *authorized views*, which are meant to enforce security policies and check user credentials [Mot89, RMSR04]. This type of access control is provided in particular by the so-called "non-Truman" access control model [RMSR04], in which the only allowed queries are those that are equivalent to authorized views or a combination thereof. The difference with respect to the previous scenario is that the system does not actually need to build a rewriting, as it will run the original query, provided that support holds.

Authorized views may be parameterized. For example, a security policy may require that a physician access a patient record only after providing the corresponding record identifier (see Example 4.1.2). In the so-called non-Truman access control model [RMSR04], a user query is considered legal only if it has an equivalent rewriting based on authorized views, or, in our terminology, if it is supported. Illegal queries are rejected by the source.

EXAMPLE 4.1.2. *Consider a source for medical data, which grants access to patient records only under some conditions. The source conforms to the following schema (where the recordNumber attribute refers to patient visit record number):*

mrecord(patientId, recordNumber)
visit(symptoms, diagnosis, recordNumber)
nextVisit(vID, vID')

and assume that recordNumber is a key for the visit relation.

A physician may have access to a limited amount of information concerning patients whose medical records belong to other colleagues, as described by the policy:

"A physician can access the diagnosis for patients only as follows. (1) He can obtain the diagnosis provided he knows the patient identifier and the visit record number. (2) He can also access the diagnosis of visits for patients with symptoms

similar to those of a patient whose id and visit record number he knows, as well as of any other follow-up visits. (3) However, the physician can access neither the patient id, nor the visit number for the visits from (2).”

Parts (1) and (2) of the policy could be implemented by separate services, whose authorized views are represented by expansions of distinguished IDB predicate ans_1 *and* ans_2 *respectively in program* $\mathcal{P}'$ *below (the ? annotation denotes parameters):*

$$
\begin{aligned}
ans_1(S, D) &:- mrecord(?N, ?R), visit(S, D, ?R),\\
ans_2(D) &:- mrecord(?N, ?R), visit(S, D', ?R),\\
&\quad\ ind_1(S, D, R')\\
ind_1(S, D, R) &:- visit(S, D', R), ind_2(D, R)\\
ind_1(S, D, R) &:- visit(S, D, R)\\
ind_2(D, R) &:- nextVisit(R, R'), ind_2(D, R')\\
ind_2(D, R) &:- visit(S, D, R)
\end{aligned}
$$

The physician wants to find the symptoms for the visit with record number r_1 *of a patient identified by* pid_1*, together with the diagnosis* D_1 *for any visit with similar symptoms, and the diagnosis* D_2 *for a subsequent visit. A conjunctive query that is supported by* $\mathcal{P}'$ *and provides the information needed is* q' *below. The primary key constraint is needed to validate the authorization because otherwise there would be no correlation between the information about symptoms used by the views witnessing support.*

$$
\begin{aligned}
q'(S, D_1, D_2) &:- mrecord(pid_1, r_1), visit(S, D_0, r_1),\\
&\quad\ visit(S, D_1, R_1), visit(S, D_1', R_1'),\\
&\quad\ nextVisit(R_1', R_2), visit(S_2, D_2, R_2)
\end{aligned}
$$

Note that the system rejects any query trying to retrieve patient ids or visit record numbers, conforming to part (3) of the policy.

Contributions. In this chapter, we carry out the (to the best of our knowledge) first study of the problems of expressibility and support under source

constraints. In particular, our contributions include:

Most permissive restrictions for decidability. We identify practically relevant restrictions on the program which ensure decidability under a mix of key and weakly acyclic foreign key constraints and beyond. The restrictions are particularly useful as they enable decidability via a reduction to the constraint-free case, which allows one to modularly "plug in" any existing algorithm to this end (such as those in [LRU99, VP97, VP00] or the one we propose here for an improved upper bound). We show that these restrictions are as permissive as possible, since their slightest relaxation leads to undecidability in the presence of even a single key constraint. This result is counter-intuitive, since the existence of a rewriting of a conjunctive query using a finite set of non-parameterized conjunctive query views under key constraints (and beyond) is known to be decidable in NP.

A widely-applicable sound test. It is unsatisfactory in practice to refuse to test support and expressibility when the decidability restrictions are violated. A more useful approach consists in devising an algorithm which functions as a decision procedure under these restrictions, yielding only a best-effort "approximation" otherwise. One pragmatic articulation of what "approximation" could mean in this context is the following: the algorithm should be *sound* (i.e. no false positives) yet it may return false negatives (i.e. is not *complete*) for inputs that do not obey the decidability restrictions. We present such an algorithm for both expressibility and support, applicable to arbitrary programs under weakly acyclic sets of embedded dependencies [AHV95], which are sufficiently expressive to capture key and foreign key constraints and beyond. The algorithm runs in deterministic exponential time in the size of the query, the size of the program and the maximum size of a constraint, which is as good as the best algorithm for rewriting queries using a finite list of views.

As a side-effect of our investigation, we *settle two open problems* left from prior work in the constraint-free setting.

Improved, practically tight upper bounds. We improve the previously best known upper bounds for deciding support in the constraint-free case: from *non-deterministic exponential* time in [VP00] and *doubly-exponential* time in [LRU99], to *deterministic exponential* time in combined query and program size. Notice that in a practical implementation, the non-deterministic exponential time upper bound of [VP00] would still result in a doubly-exponential algorithm. The improvement is achieved using the sound algorithm mentioned above, which provably acts as an exponential-time decision procedure in the absence of constraints. We show our algorithm to be optimal in the program size (we give a deterministic EXPTIME lower bound for fixed query) and optimal for practical purposes in the query size (we give an NP lower bound for fixed program). The question of the tightness of this NP lower bound remains open. An interesting consequence of our new upper bound is that, in practical implementations, rewriting using an infinite set of views is no more expensive than using finitely many views listed individually (still deterministic exponential time).

The relationship between expressibility and support. We establish that expressibility and support are inter-reducible in PTIME in both the absence and the presence of constraints. This enables us to characterize the complexity of expressibility as well, and to employ the same algorithm for solving both problems. The result comes as a pleasant surprise, since prior work reports distinct upper bounds for these problems, suggesting (in line with intuition) that finding a rewriting of the query using program expansions is harder than finding a single equivalent expansion.

A one-size-fits-all solution. It is remarkable (and practically appealing) that all our upper bound results are based on the *same* algorithm for support, which serves simultaneously as (i) an essentially optimal decision procedure in the constraint-free case, improving prior upper bounds, (ii) a decision procedure under constraints in all known decidable cases, (iii) a sound procedure in general, and (iv)

all of the above for the problem of expressibility, due to our inter-reducibility result.

Chapter outline. After introducing preliminary concepts, results and notation in Section 4.2, in Section 4.3 we establish the PTIME inter-reducibility of expressibility and support. Section 4.4 presents decidable restrictions and Section 4.5 contains a sound algorithm in the case of general constraints. We also show there the improved upper bounds for the constraint-free setting (Section 4.5.1). For presentation simplicity, throughout these sections we ignore the presence of parameters in the views generated by the program. We map the boundaries of decidability in Section 4.6 and show how parameters are handled in Section 4.7. Finally, we discuss related work in Section 4.8.

4.2 Preliminaries

We denote with CQ the language of conjunctive queries.

Constraints. We consider constraints ξ of the form

$$\forall\bar{u}\forall\bar{w}\ \phi(\bar{u},\bar{w}) \longrightarrow \exists\bar{v}\,\psi(\bar{u},\bar{v})$$

where ϕ (the *premise*) and ψ (the *conclusion*) are conjunctions of relational or equality atoms. Such constraints are known as *embedded dependencies* and are sufficiently expressive to specify all usual integrity constraints, such as keys, foreign keys, inclusion, join, multivalued dependencies, EGDs, TGDs etc. [AHV95]. We call ϕ the *premise* and ψ the *conclusion*. If $\bar{v}$ is empty, then ξ is a *full dependency*. If ψ consists only of equality atoms, then ξ is an *equality-generating dependency (EGD)*. If ψ consists only of relational atoms, then ξ is a *tuple-generating dependency (TGD)*. If the premise and conclusion of a TGD contain one atom each, we call it an *inclusion dependency (IND)*. An IND in which the variables $\bar{u}$ appear precisely in the key attributes of the relation mentioned in the conclusion is a *foreign key* constraint. A *key constraint* on relation R can be expressed by the EGD $\forall\bar{u},\bar{v}_1,\bar{v}_2\ R(\bar{u},\bar{v}_1)\wedge R(\bar{u},\bar{v}_2) \longrightarrow \bar{v}_1=\bar{v}_2$. We write $A \models \mathcal{C}$ if the instance A satisfies all the constraints in $\mathcal{C}$.

Containment and Equivalence. Query Q_1 is contained in query Q_2 under the set $\mathcal{C}$ of constraints (denoted $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$) iff $Q_1(D) \subseteq Q_2(D)$ for every database $D \models \mathcal{C}$, where $Q(D)$ denotes the result of Q on D. Q_1 is equivalent to Q_2 under $\mathcal{C}$ (denoted $Q_1 \equiv_{\mathcal{C}} Q_2$) iff $Q_1 \sqsubseteq_{\mathcal{C}} Q_2$ and $Q_2 \sqsubseteq_{\mathcal{C}} Q_1$.

Mappings. A *partial mapping* from CQ query Q_1 to CQ query Q_2 is a function h from the variables and constants of Q_1 to the variables and constants of Q_2 such that (i) h is the identity mapping on all constants, and (ii) for every relational atom (also called subgoal) $R(\bar{X})$ of Q_1, if h is defined for all variables in $(\bar{X})$, then $R(h(\bar{X}))$ is a subgoal of Q_2. A *homomorphism* from a set of subgoals C_1 to a set of subgoals C_2 is a partial mapping from the query $Q_1() :- C_1$ to the query $Q_2() :- C_2$ which is defined on all variables of Q_1. A *containment mapping* from CQ query Q_1 with tuple of head variables $\bar{X}_1$ to CQ query Q_2 with tuple of head variables $\bar{X}_2$ is a homomorphism h from Q_1 to Q_2 such that $h(\bar{X}_1) = \bar{X}_2$. We represent mappings as sets of pairs associating variables with either variables or constants, and use the notation $X : Y$ for the pair (X, Y). The *union* of two mappings is simply the union of their sets of pairs. A mapping is *consistent* if it does not map the same variable to two distinct values. A set of mappings is *compatible* if their union is consistent. Composition of mappings is the standard function composition, denoted by the operator $\circ$.

Expansion using views. Given a CQ query R formulated in terms of a set of view names $\mathcal{V}$(where the views are also CQs), the *expansion* of query R w.r.t. the views in $\mathcal{V}$ (denoted $expand_{\mathcal{V}}(R)$) is the query E obtained as follows: every subgoal $V(\bar{X})$ in R is replaced by a copy of the body of V, in which the head variables of V are renamed to $\bar{X}$ and all other variables are replaced by variables occurring in no other view bodies introduced during the expansion. It is easy to see that this variable renaming defines a homomorphism h from V into the expansion E, which we refer to as the *expansion homomorphism*.

Rewriting using views. We say that a conjunctive query R formulated in terms of view names $\mathcal{V}$ is a rewriting of a query Q using $\mathcal{V}$ under a set $\mathcal{C}$ of dependencies iff $Q \equiv_{\mathcal{C}} expand_{\mathcal{V}}(R)$.

Equivalence under views and constraints. Given queries R_1, R_2 formulated in terms of the view names in $\mathcal{V}$ and a set of dependencies $\mathcal{C}$, we say that R_1 is equivalent to R_2 under $\mathcal{V}$ and $\mathcal{C}$, denoted $R_1 \equiv_{\mathcal{C}}^{\mathcal{V}} R_2$, if and only if $expand_{\mathcal{V}}(R_1) \equiv_{\mathcal{C}} expand_{\mathcal{V}}(R_2)$.

The chase. We will use the classical *chase* procedure for rewriting conjunctive queries using a set of embedded dependencies [AHV95]. For arbitrary sets $\mathcal{C}$ of dependencies, the chase is not guaranteed to terminate. The least restrictive condition on $\mathcal{C}$ known to date which is sufficient to ensure termination of the chase with $\mathcal{C}$ regardless of the query Q is called *weak acyclicity* [FKMP03] (see also [DT03b]). Weak acyclicity of $\mathcal{C}$ implies termination of the chase of Q with $\mathcal{C}$ in time polynomial in the size of Q and exponential in the size of $\mathcal{C}$. Assuming termination of the chase, we denote with $chase_{\mathcal{C}}(Q)$ the query obtained by chasing conjunctive query Q with $\mathcal{C}$ to termination (this query is unique up to equivalence). Besides introducing new variables (for instance due to chasing with TGDs), the chase may equate the original variables of Q to constants or to each other (for instance due to chasing with key constraints) [AHV95]. Denoting this variable renaming with r, it is a well-known fact that r is a homomorphic mapping from Q into $chase_{\mathcal{C}}(Q)$, also called the *chase homomorphism* [AHV95].

Datalog expansions. A finite expansion (in short "expansion") of an IDB predicate p of a Datalog program $\mathcal{P}$ is a CQ query with head $p(\bar{X})$ and body obtained as follows: initialize the body to $body := p(\bar{X})$, then apply the following expansion step a finite number of times until no more IDBs are left in the body: for every IDB goal g_i in the body, pick a rule r_i in $\mathcal{P}$ defining g_i and collect all picked rules in a list $\mathcal{V}$. Treating $\mathcal{V}$ as views, replace *body* with $expand_{\mathcal{V}}(body)$, where each g_i is expanded using r_i. The set of expansions of $\mathcal{P}$ is infinite if $\mathcal{P}$ is recursive.

Convention. *In the remainder of this chapter, unless explicitly stated otherwise, all queries and views are conjunctive queries, all programs are Datalog programs, and all dependencies are embedded dependencies.*

4.3 Expressibility Versus Support

We say that a view V is *generated* by program $\mathcal{P}$ when V is a CQ expansion of $\mathcal{P}$.

Definition 4.3.1. *Given a Datalog program $\mathcal{P}$, a conjunctive query Q and a set of embedded dependencies $\mathcal{C}$, we say that*

1. *Q is* supported *by $\mathcal{P}$ under $\mathcal{C}$ (denoted $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$), iff there is a* finite *set of views $\mathcal{V}$ generated by $\mathcal{P}$ and a conjunctive query rewriting of Q using $\mathcal{V}$ under $\mathcal{C}$.*

2. *Q is* expressible *by $\mathcal{P}$ under $\mathcal{C}$ (denoted $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$), iff Q is equivalent under $\mathcal{C}$ to some view V generated by $\mathcal{P}$.*

In previous work, the problems of support and expressibility were introduced separately (in [LRU99], respectively [VP00]). They were shown to be decidable, yet their reported complexity upper bounds were different even in the absence of constraints: doubly-exponential deterministic time for support [LRU99], and EXPTIME for expressibility [VP00]. These results seemed to follow the intuition that finding a rewriting of the query using some expansions of the program is harder than finding a single equivalent expansion.

We establish a counter-intuitive relationship between the two problems, showing them to be inter-reducible in polynomial time even in the presence of dependencies.

Theorem 4.3.1. *Let $\mathcal{C}$ be a weakly acyclic set of embedded dependencies. Then there is a reduction from the problem of support of a query Q by a program $\mathcal{P}$ under $\mathcal{C}$ to an instance of the expressibility problem, which is in PTIME in the size of Q and $\mathcal{P}$ and in EXPTIME in the size of $\mathcal{C}$.*

Given Q, $\mathcal{P}$ and $\mathcal{C}$, we construct a new query Q', program $\mathcal{P}'$ and set of dependencies $\mathcal{C}'$, such that Q is supported by $\mathcal{P}$ under $\mathcal{C}$ iff Q' is expressible by $\mathcal{P}'$ under $\mathcal{C}'$.

The reduction starts from the following result, which generalizes a result of [LMSS95b] to the presence of dependencies:

Lemma 4.3.1. *Let $\mathcal{C}$ be a weakly acyclic set of embedded dependencies. Then* $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *holds iff there is a rewriting R of Q under $\mathcal{C}$ using views generated by $\mathcal{P}$, where R has no more variables than $chase_{\mathcal{C}}(Q)$.*

Proof: [of Theorem 4.3.1] It was shown in prior work [DLN05] that, if $\mathcal{C}$ is weakly acyclic, then $chase_{\mathcal{C}}(Q)$ contains v variables, where v is upper-bounded by a polynomial in the number of goals in Q and exponential in the maximum arity of a relation appearing in the conclusion of a dependency in $\mathcal{C}$.

From this, we will build in PTIME in the size of $chase_{\mathcal{C}}(Q)$ and $\mathcal{P}$ a new program $\mathcal{P}'$ that basically enumerates all possible conjunctions of expansions of $\mathcal{P}$using at most v variables.

For this proof, it helps to consider Q and conjunctions of expansions as *rectified*. More precisely, no constants are allowed in predicate subgoals, and no variable appears twice in subgoals. Instead, joins are made explicit by subgoals $equals(X, Y)$, and selections with a constant c by subgoals $equals(X, c)$. Note that we can pass from any conjunctive query to its rectified version and vice-versa in linear time.

Given Q, denote with a_Q the arity of Q (the number of its distinguished variables). Assume w.l.o.g. that the distinguished predicate of $\mathcal{P}$ is ans, of arity $a_{\mathcal{P}}$. We add a new IDB predicate ans', as well as a new unary EDB predicate D.

We build the following program, of distinguished predicate ans':

$$\begin{aligned}
ans'(V_1,\ldots,V_{a_Q}) \;&:-\; pick(V_1,X_1,\ldots,X_v),\\
&\quad\; pick(V_2,X_1,\ldots,X_v),\ldots\\
&\quad\; pick(V_{a_Q},X_1,\ldots,X_v),\\
&\quad\; temp(X_1,\ldots,X_v)\\
temp(X_1,\ldots,X_v) \;&:-\; D(X_1),\ldots,D(X_v)\\
temp(X_1,\ldots,X_v) \;&:-\; ans(Y_1,Y_2,\ldots,Y_{a_\mathcal{P}}),\\
&\quad\; pick(Y_1,X_1,\ldots,X_v),\\
&\quad\; pick(Y_2,X_1,\ldots,X_v),\ldots\\
&\quad\; pick(Y_{a_\mathcal{P}},X_1,\ldots,X_v),\\
&\quad\; temp(X_1,\ldots,X_v)
\end{aligned}$$

$$\begin{aligned}
pick(V,X_1,\ldots,X_v) \;&:-\; equals(V,X_1),\\
&\quad\; D(V),D(X_1),\ldots,D(X_v)\\
pick(V,X_1,\ldots,X_v) \;&:-\; equals(V,X_2),\\
&\quad\; D(V),D(X_1),\ldots,D(X_v)\\
&\vdots\\
pick(V,X_1,\ldots,X_v) \;&:-\; equals(V,X_v),\\
&\quad\; D(V),D(X_1),\ldots,D(X_v)
\end{aligned}$$

$$+$$

modified rules of $\mathcal{P}$ *with* D *atoms for all variables*

The rules of $\mathcal{P}$ appear in $\mathcal{P}'$ modified as follows. Each rule of $\mathcal{P}$ of the form

$$head_i(\bar{X}_i) :- body_i(\bar{X}_i,\bar{Y}_i)$$

is transformed into a rule

$$\begin{aligned}
head_i(\bar{X}_i) :-\; & body_i(\bar{X}_i,\bar{Y}_i),\\
& D(X_1),\ldots,D(X_{n_i}),D(Y_1),\ldots,D(Y_{m_i})
\end{aligned}$$

The rules added in addition to those of $\mathcal{P}$ have the task of expressing all possible conjunctive queries with a_Q head variables and at most v variables in total, formulated against the distinguished goal of $\mathcal{P}$, *ans*. The *ans* subgoals are then expanded into views generated by $\mathcal{P}$ (plus D subgoals), due to the inclusion of the (modified) rules of $\mathcal{P}$ into $\mathcal{P}'$.

Note that the *temp* subgoal lists the pool of v variables the expansions of $\mathcal{P}'$ will use. Each *temp* subgoal expands into arbitrarily many *ans* subgoals which will build the body of the rewriting. The variables appearing in the head *ans'* and in the various *ans* subgoals in the body are each associated with *pick* subgoals. The *pick* subgoal has v possible expansions, each having the role of picking one of the v variables in the pool to equate with the variable in its first argument. In this way, every assignment of variables from the pool to variables of (the head and body of) the conjunctive query over *ans* subgoals is realizable by some expansion of the *pick* subgoals.

The D predicate is introduced for technical purposes, to avoid generating unsafe Datalog rules for the *pick* goal. Its effect is that each view generated by $\mathcal{P}'$ has a D subgoal for each of its variables. This does not influence expressibility as long as we add such subgoals for all variables appearing in the query and in the dependencies. Indeed, if Q has the form

$$Q(Z_1, \ldots, Z_{a_Q}) \quad :- \quad body(Z_1, \ldots, Z_{v_Q})$$

with *body* a conjunction of subgoals, we build a new boolean query

$$\begin{aligned} Q'(Z_1, \ldots, Z_{a_Q}) \quad :- \quad & body(Z_1, \ldots, Z_{v_Q}), \\ & D(Z_1), \ldots, D(Z_{v_Q}) \end{aligned}$$

Finally, we construct a new set of dependencies $\mathcal{C}'$ by adding in the conclusion of each dependency σ from $\mathcal{C}$ the predicate $D(X)$ for every variable X appearing in σ.

Notice that $\mathcal{C}'$ and Q' are obtained in linear time from $\mathcal{C}$ and Q, respectively. $\mathcal{P}'$ is obtained in PTIME from $\mathcal{P}$ and v, where the latter is polynomial in the size of Q but exponential in the maximum arity of a relation appearing in the conclusion of some dependency from $\mathcal{C}$.

It is now easy to show that $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds if and only if $\text{EXPR}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$ does.

We consider the "if" direction. Assuming that $\text{EXPR}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$ holds, let V denote some expansion of $\mathcal{P}'$ such that $V \equiv_{\mathcal{C}'} Q'$. First, it is easy to check that for the query V' obtained from V by removing all D subgoals we have $V' \equiv_C Q$. Note also that V was generated by first expanding an ans' goal into a conjunction of ans and D subgoals (via some intermediary *pick* and *temp* steps), and then expanding each ans subgoal using the modified versions of $\mathcal{P}$'s rules (adding a D subgoal for each new variable). Therefore, its V' part is isomorphic with the expansion of some conjunctive query over $\mathcal{P}$ expansions. Hence V' represents a witness for $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

The "only if" direction is similar. Recall that ans' goals can expand into any conjunction of ans subgoals using at most v variables (plus the corresponding D subgoals). Therefore, any rewriting R of Q over views generated by $\mathcal{P}$ and using at most v variables expands into a query that is isomorphic with some expansion of $\mathcal{P}'$, modulo missing D subgoals. Such a rewriting R would thus be a witness for $\text{EXPR}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$. Since by Lemma 4.3.1 we can apply this upper-bound on the number of variables in a rewriting, we can conclude that $\text{SUPP}^{\mathcal{C}}_{\mathcal{P}}(Q)$ implies $\text{EXPR}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$. ■

Corollary 4.3.1. *If the size of the schema (with dependencies) is bounded by a constant, then there is a PTIME reduction from support to expressibility provided the set of embedded dependencies is weakly acyclic.*

Corollary 4.3.2. *In the absence of dependencies, there is a PTIME reduction from support to expressibility.*

The next result shows the existence of a polynomial-time reduction in the other direction, requiring no restrictions on the embedded dependencies.

Theorem 4.3.2. *Expressibility reduces in PTIME to support.*

Proof: Given Q,$\mathcal{P}$ and $\mathcal{C}$, we construct a boolean query Q'', boolean program $\mathcal{P}''$ and set of dependencies $\mathcal{C}''$, such that Q is expressible by $\mathcal{P}$ under $\mathcal{C}$ iff Q'' is supported by $\mathcal{P}''$ under $\mathcal{C}''$.

For presentation simplicity, we first show a first-cut solution which works only if the query graph is connected, then we explain how the reduction can be adapted to arbitrary queries.

Denote with a_Q the arity of Q and assume w.l.o.g. that Q has the form

$$Q(Z_1, \ldots, Z_{a_Q}) \quad :- \quad body(Z_1, \ldots, Z_{v_Q})$$

with *body* a conjunction of subgoals and $v_Q \geq a_Q$ the total number of variables appearing in Q. We build the boolean query

$$Q'() \quad :- \quad head(Z_1, \ldots, Z_{a_Q}), body(Z_1, \ldots, Z_{v_Q})$$

using a fresh EDB relation *head* of arity a_Q.

Assume w.l.o.g. that the distinguished IDB of $\mathcal{P}$ is *ans*. Notice that, for Q to be expressible by $\mathcal{P}$, *ans* must have the same arity as Q. $\mathcal{P}'$ is constructed by adding to the rules of $\mathcal{P}$ a new rule defining a fresh, boolean IDB predicate ans':

$$ans'() \quad :- \quad ans(X_1, \ldots, X_{a_Q}), head(X_1, \ldots, X_{a_Q}).$$

The distinguished IDB predicate of $\mathcal{P}'$ is ans'.

Note that the views generated by $\mathcal{P}'$ are in one-to-one correspondence to those generated by $\mathcal{P}$: any view V' generated by $\mathcal{P}'$ simply extends the body of some view V generated by $\mathcal{P}$ with a *head* subgoal containing the head variables of V. Q is equivalent to V if and only if Q' is equivalent to the corresponding view V': the *head* subgoals appearing in both Q' and V' ensure the desired correspondence between the distinguished variables of Q and those of V. We have thus proven

Claim 1. $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

Also note that, since each view generated by $\mathcal{P}'$ is boolean, any rewriting using such views is really a Cartesian product thereof. We therefore make the following claim:

Claim 2. Consider a boolean query Q'' and the set of embedded dependencies $\mathcal{C}''$. If

(a) Q'' performs no Cartesian products (i.e. if its hypergraph [AHV95] is connected), and
(b) all constraints in $\mathcal{C}''$ have premises with connected hypergraph,

then Q'' is equivalent under $\mathcal{C}''$ to some boolean conjunctive query R iff it is equivalent under $\mathcal{C}''$ to a connected subquery of R. $\diamond$

Proof of Claim 2. The "if" direction is immediate, we prove the "only if" direction next.

Let Q'' be connected, and $R() :- \ V_1(), V_2()$, where the hypergraphs of V_1 and V_2 are disjoint.

Assume toward a contradiction that $V_1 \not\sqsubseteq_{\mathcal{C}''} V_2$ and $V_2 \not\sqsubseteq_{\mathcal{C}''} V_1$. Then there must exist two databases, DB_1, DB_2, with disjoint active domains, such that both DB_1, DB_2 satisfy $\mathcal{C}''$, and such that $V_1(DB_1) = \mathit{true}$, $V_2(DB_1) = \mathit{false}$, $V_1(DB_2) = \mathit{false}$ and $V_2(DB_2) = \mathit{true}$. Since Q'' is equivalent to R under $\mathcal{C}''$, we obtain that $Q(DB_1) = Q(DB_2) = \mathit{false}$.

Let DB_3 be the database obtained by unioning the two:
$DB_3 := DB_1 \cup DB_2$.

We claim that DB_3 satisfies $\mathcal{C}''$ as well: the components DB_1, DB_2 do so by hypothesis, and their disjoint union cannot violate any constraint in $\mathcal{C}''$ because all constraint premises are connected and thus cannot match across the databases.

Note that $Q''(DB_3) = \mathit{false}$, as Q'' has no match into any of DB_1, DB_2, and no match across them because it is connected. Also note that $R(DB_3) = \mathit{true}$, as V_1 and V_2 have a match against the sub-databases DB_1 and DB_2, respectively.

We have thus exhibited a database $DB_3 \models \mathcal{C}''$ such that $R(DB_3) = \mathit{true}$, but $Q''(DB_3) = \mathit{false}$, contradicting the equivalence of Q to R under $\mathcal{C}''$. Therefore, either of V_1, V_2 must be contained in the other under $\mathcal{C}''$, so R can be minimized under $\mathcal{C}''$ to just one component. The reasoning extends to arbitrarily many components by induction.

End of proof of Claim 2.

By Claim 2, we have that, under restrictions (a) and (b), all rewritings of Q' under $\mathcal{C}$ using views generated by $\mathcal{P}'$ contain a single view goal or can be minimized to a single view goal. This implies that $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ holds if and only if $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ does. Considering also Claim 1, we obtain $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

We now refine the reduction, lifting restrictions (a) and (b). To this end, we obtain from $Q', \mathcal{P}'$ and $\mathcal{C}$, $Q'', \mathcal{P}''$ and $\mathcal{C}$" such that $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff $\textsc{Expr}^{\mathcal{C}''}_{\mathcal{P}''}(Q'')$ does, and such that Q'' and the premises of all constraints in $\mathcal{C}''$ are connected. Then Claim 2 will apply to Q'' and $\mathcal{C}''$, completing the proof.

The head of Q'' is the same as that of Q'. The distinguished IDB of $\mathcal{P}''$ is the same as that of $\mathcal{P}'$. Every remaining goal and subgoal of Q' and $\mathcal{P}'$, say $G(\bar{X})$ of arity a, is extended to an $a+1$-ary goal $G(\bar{X}, U)$, where U is a fresh variable shared across all goals.

We replace in the same way all subgoals appearing in dependencies in $\mathcal{C}$: for every $\sigma \in \mathcal{C}$ of form

$$\forall \bar{X}\ \mathit{premise}(\bar{X}) \rightarrow \exists \bar{Y}\ \mathit{conclusion}(\bar{X}, \bar{Y}),$$

we construct σ'' of form

$$\forall \bar{X} \forall U\ \mathit{premise}''(\bar{X}, U) \rightarrow \exists \bar{Y}\ \mathit{conclusion}''(\bar{X}, \bar{Y}, U),$$

where *premise"* and *conclusion"* are obtained from *premise* and *conclusion*, respectively, by extending the goals with the new variable U, as done above for Q' and $\mathcal{P}'$.

Claim 3. $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}'}(Q')$ holds iff $\textsc{Expr}^{\mathcal{C}''}_{\mathcal{P}''}(Q'')$ does.

The theorem follows from Claims 1, 3 and 2. ■

In particular, since dependency-free support is known to be decidable [LRU99], Theorem 4.3.2 implies decidability of dependency-free expressibility, with the same complexity.

4.4 Decidable Cases

In this section, we give restrictions under which the problems of expressibility and support are decidable under constraints. As will be seen in Section 4.6, the restrictions are needed because the two problems are in general undecidable, and they are fairly tight, in the sense that even slight relaxations thereof lead to undecidability.

Because it is interesting in its own right, we show a particular route to decidability based on reducing to the dependency-free setting, which is known to be decidable [LRU99]. However, this does not yet provide the improved upper bound, which requires improving prior results for the dependency-free case. We shall do so in Section 4.5, obtaining a more general result: a novel algorithm that does not rely on reduction to the dependency-free case, but serves as an optimal decision procedure when dependencies are absent or when they satisfy the restrictions presented in this section, and gracefully degenerates to a sound procedure otherwise.

We introduce properties of the program and of the views it generates that suffice for our reduction to the dependency-free case. The idea is to pre-process the program to explicitly incorporate into it the knowledge about the dependencies, so that these can then be ignored, thus reducing the problem to dependency-free expressibility and support for the new program. The pre-processing technique relies on the *chase* procedure. This was a natural choice, as the chase tool has been traditionally employed successfully to reduce classical decision problems (such as query equivalence or implication of dependencies [AHV95]) from the presence of dependencies to their absence. We start with expressibility.

Given a Datalog program $\mathcal{P}$, we denote with $chase_{\mathcal{C}}(\mathcal{P})$ the program obtained by chasing each rule of $\mathcal{P}$ with $\mathcal{C}$.

Definition 4.4.1 ($\mathcal{C}$-Local Program)**.** *Let $\mathcal{C}$ be a weakly acyclic set of dependencies. We say that a program $\mathcal{P}$ is* $\mathcal{C}$-local *iff for every view V generated by $\mathcal{P}$ there is a view W generated by $chase_{\mathcal{C}}(\mathcal{P})$, and for every view W generated by $chase_{\mathcal{C}}(\mathcal{P})$ there is a view V generated by $\mathcal{P}$, such that $chase_{\mathcal{C}}(V)$ is equivalent to W even in the absence of dependencies.*

The intuition behind $\mathcal{C}$-locality is as follows. Recall that when checking expressibility under $\mathcal{C}$, one needs to exhibit some view V generated by $\mathcal{P}$, such that $Q \equiv_{\mathcal{C}} V$. By the chase theorem [AHV95, MMS79], if the chase terminates, the equivalence under $\mathcal{C}$ reduces to the following equivalence in the absence of dependencies (i.e. under the empty set of dependencies): $chase_{\mathcal{C}}(Q) \equiv_{\emptyset} chase_{\mathcal{C}}(V)$. $\mathcal{C}$-locality ensures that the chase of view V can be avoided by simply searching among the views generated by $chase_{\mathcal{C}}(\mathcal{P})$. These must include some W with $W \equiv_{\emptyset} chase_{\mathcal{C}}(V)$, so the existence of V as above is equivalent to the existence of W generated by $chase_{\mathcal{C}}(\mathcal{P})$, with $chase_{\mathcal{C}}(Q) \equiv_{\emptyset} W$. This in turn is by definition dependency-free expressibility of query $chase_{\mathcal{C}}(Q)$ by program $chase_{\mathcal{C}}(\mathcal{P})$. Indeed, we can show the following.

Theorem 4.4.1. *Let Q be a conjunctive query, $\mathcal{C}$ a weakly acyclic set of dependencies, and $\mathcal{P}$ a $\mathcal{C}$-local program. Then $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff $\textsc{Expr}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ holds.*

Proof: Let V be the view generated by $\mathcal{P}$, witnessing $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$, i.e.

$$Q \equiv_{\mathcal{C}} V \tag{4.2}$$

Because $\mathcal{C}$ is weakly acyclic, the chase with it is guaranteed to terminate, so (4.2) is equivalent to

$$chase_{\mathcal{C}}(Q) \equiv chase_{\mathcal{C}}(V) \tag{4.3}$$

Since $\mathcal{P}$ is $\mathcal{C}$-local, there is W generated by $chase_{\mathcal{C}}(\mathcal{P})$ with

$$chase_{\mathcal{C}}(V) \equiv W. \tag{4.4}$$

By (4.4) and (4.3) and by transitivity of $\equiv$ relation, we obtain

$$chase(Q) \equiv W \tag{4.5}$$

and thus W witnesses $\textsc{Expr}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

The opposite direction is analogous. ■

The reduction of support to the dependency-free case requires an additional restriction on the views generated by the program. In this case, we need to exhibit a set $\mathcal{V}$ of views generated by $\mathcal{P}$ and a rewriting R of Q in terms of $\mathcal{V}$. Again by the chase theorem [AHV95, MMS79], this is equivalent (provided the chase terminates) to exhibiting $\mathcal{V}$ and R such that $chase_{\mathcal{C}}(Q) \equiv_{\emptyset} chase_{\mathcal{C}}(expand_{\mathcal{V}}(R))$. The idea behind the reduction is to require the views to be such that no matter how they are used in R, chasing R's expansion gives the same result as first chasing each view individually and then expanding R with the chased views: $chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)) \equiv_{\emptyset} expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R)$. Now if $\mathcal{P}$ is $\mathcal{C}$-local, then the chased views are equivalent to some views $\mathcal{W} = \{W_1, \ldots, W_n\}$ generated by $chase_{\mathcal{C}}(\mathcal{P})$, and we have $chase_{\mathcal{C}}(Q) \equiv_{\emptyset} chase_{\mathcal{C}}(expand_{\mathcal{W}}(R))$, which is the definition of dependency-free support of $chase_{\mathcal{C}}(Q)$ by $chase_{\mathcal{C}}(P)$. We formalize this intuition next.

Definition 4.4.2 ($\mathcal{C}$-Independent View Set)**.** *Let $\mathcal{C}$ be a weakly acyclic set of dependencies. We say that a set of views $\mathcal{V} = \{V_1, \ldots, V_n\}$ is* $\mathcal{C}$-independent *iff, for every query R' formulated in terms of $\mathcal{V}$, there exists query R also formulated in terms of $\mathcal{V}$, such that*

(i) $R' \equiv_{\mathcal{C}}^{\mathcal{V}} R$,

(ii) and such that

$$chase_{\mathcal{C}}(expand_{\{V_1,\ldots,V_n\}}(R))$$

is equivalent even in the absence of dependencies to

$$expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R).$$

Notice that we do not require property (ii) in Definition 4.4.2 to hold for all queries R' over $\mathcal{V}$, since there are potentially many equivalent forms of R'. It suffices if one of them satisfies (ii). In that case, we can show the following.

Theorem 4.4.2. *Let Q be a conjunctive query, $\mathcal{C}$ a weakly acyclic set of dependencies, and $\mathcal{P}$ a $\mathcal{C}$-local program. Then, if the views generated by $\mathcal{P}$ are $\mathcal{C}$-independent, then* $\textsc{Supp}_{\mathcal{P}}^{\mathcal{C}}(Q)$ *iff* $\textsc{Supp}_{chase_{\mathcal{C}}(\mathcal{P})}^{\emptyset}(chase_{\mathcal{C}}(Q))$.

Proof: Assume w.l.o.g. that $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ is witnessed by the views $\mathcal{V} = \{V_1, \ldots, V_n\}$ generated by $\mathcal{P}$ and the rewriting R in terms of $\mathcal{V}$. By $\mathcal{C}$-locality of $\mathcal{P}$, there are $W_1, \ldots, W_n$ generated by $chase_{\mathcal{C}}(\mathcal{P})$ such that $W_i \equiv chase_{\mathcal{C}}(V_i)$ for every $1 \leq i \leq n$. We therefore have:

$$Q \equiv_{\mathcal{C}} expand_{\mathcal{V}}(R), \tag{4.6}$$

which is equivalent (due to weak acyclicity of $\mathcal{C}$) to

$$chase_{\mathcal{C}}(Q) \equiv chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)), \tag{4.7}$$

By $\mathcal{C}$-independence of $\mathcal{V}$ there must be another query R' over the view schema such that

$$chase_{\mathcal{C}}(expand_{\mathcal{V}}(R')) \equiv expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R') \tag{4.8}$$

and

$$expand_{\mathcal{V}}(R') \equiv_{\mathcal{C}} expand_{\mathcal{V}}(R) \tag{4.9}$$

hence again by weak acyclicity of $\mathcal{C}$,

$$chase_{\mathcal{C}}(expand_{\mathcal{V}}(R')) \equiv chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)) \equiv chase_{\mathcal{C}}(Q) \tag{4.10}$$

From (4.8) and (4.10), we infer:

$$chase_{\mathcal{C}}(Q) \equiv expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R'), \tag{4.11}$$

which is equivalent to

$$chase_{\mathcal{C}}(Q) \equiv expand_{\{W_1,\ldots,W_n\}}(R') \tag{4.12}$$

because the semantics of a query composition is preserved under replacement with equivalent queries.

But then $\{W_1, \ldots, W_n\}$ and R' witness

$$\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q)).$$

■

We next provide various syntactic restrictions on the dependencies in $\mathcal{C}$ and on $\mathcal{P}$ to guarantee $\mathcal{C}$-independence and $\mathcal{C}$-locality.

Theorem 4.4.3. *Let $\mathcal{C}$ be a weakly acyclic set of inclusion dependencies. Then any Datalog program $\mathcal{P}$ is $\mathcal{C}$-local and every finite subset of its generated views is $\mathcal{C}$-independent.*

Proof:

Let α be a conjunctive query whose body contains both EDB and IDB relations of $\mathcal{P}$. We say that β is obtained in one expansion step with rule r, denoted

$$\alpha \stackrel{r}{\Longrightarrow} \beta,$$

iff β is obtained by expanding with r one of α's subgoals that uses the IDB relation defined by r. Given a sequence of expansion steps

$$ans(\bar{X}) = \alpha_0 \stackrel{r_1}{\Longrightarrow} \alpha_1 \ldots \stackrel{r_n}{\Longrightarrow} \alpha_n$$

where each r_i is a rule of $\mathcal{P}$ and ans is a distinguished IDB of $\mathcal{P}$, we call each α_i a *partial expansion* of $\mathcal{P}$.

To prove the theorem, we claim more specifically that the result of chasing any partial expansion α of $\mathcal{P}$ can be alternatively obtained by replacing the rules in the derivation of α with their chased form (these are rules of $chase_{\mathcal{C}}(\mathcal{P})$):

Claim 1. For every n, and every sequence of expansion steps

$$ans(\bar{X}) = \alpha_0 \stackrel{r_1}{\Longrightarrow} \alpha_1 \ldots \stackrel{r_n}{\Longrightarrow} \alpha_n,$$

there is a sequence of expansion steps

$$ans(\bar{X}) = \beta_0 \stackrel{chase_{\mathcal{C}}(r_1)}{\Longrightarrow} \beta_1 \ldots \stackrel{chase_{\mathcal{C}}(r_n)}{\Longrightarrow} \beta_n$$

such that for every $0 \leq i \leq n$, β_i is isomorphic to $chase_{\mathcal{C}}(\alpha_i)$.

Notice that, if the claim holds, then any partial expansion α_n of $\mathcal{P}$ can be obtained (up to isomorphism) as an expansion β_n of $chase_{\mathcal{C}}(\mathcal{P})$. This immediately gives $\mathcal{C}$-locality for the particular case of full expansions (which are the generated

views): to find the view W generated by $chase_{\mathcal{C}}(\mathcal{P})$ that corresponds to view V generated by $\mathcal{P}$, retrace the derivation of V by $\mathcal{P}$ using the chased rules instead.

Proof of Claim 1. The claim is proven by induction on n, using for the induction step the observation that INDs have only one atom in the premise, so the EDB goals in α_i cannot cooperate with the new EDB goals introduced in α_{i+1} to enable a chase step. The chase of α_{i+1} therefore progresses in isolation on the goals appearing in α_i, and on the new goals introduced by the expansion step. Its effect is therefore alternatively achievable by expanding $chase_{\mathcal{C}}(\alpha_i)$ in one step using the chased rule $chase_{\mathcal{C}}(r_{i+1})$.
End of proof of Claim 1.

$\mathcal{C}$-independence of the views follows from essentially the same observation about INDs, which actually gives a stronger result:

Claim 2. Any set $\mathcal{V}$ of views (regardless of whether generated by some program or not) is $\mathcal{C}$-independent if $\mathcal{C}$ consists only of INDs.

Indeed, any join of renamed copies of view bodies (corresponding to the expansion of some rewriting), when chased, gives the same result as chasing the view bodies in isolation and then joining them. This is because the single-atom premises of the INDs preclude the interaction (w.r.t. enabling chase steps) of goals from distinct view bodies. ■

Theorems 4.4.1, 4.4.2 and 4.4.3 immediately imply that for weakly acyclic sets of inclusion dependencies, expressibility and support reduce to the dependency-free versions:

Corollary 4.4.1. *If $\mathcal{C}$ is a weakly acyclic set of inclusion dependencies, then for any program $\mathcal{P}$ and query Q,* $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Expr}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ *and* $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

EXAMPLE 4.4.1. *Consider a source for travel data using the following schema:*

train(origin, destination, operator)
bus(origin, destination,operator)

where each origin-destination pair is connected by a non-stop leg. It accepts queries for train itineraries with arbitrary many legs in which the same operator is used. It returns the origin, the destination, one intermediary stop and the operator. This family of queries is described by program $\mathcal{P}$:

$$\begin{array}{rrll} (\mathcal{P}) & ans(A,B,C,O) & :- & ind(A,B,O), ind(B,C,O) \\ & ind(B,C,O) & :- & t(B,B',O), ind(B',C,O) \\ & ind(B,C,O) & :- & t(B,C,O) \end{array}$$

Let Q be an application query searching for a one-way trip with connection in Paris, such that starting from Paris one can either continue the trip by bus, and stay with the first operator, or take another train with any available operator.

$$(Q) \quad q(A,B) :- t(A,C,O_1), b(C,B,O_1), t(C,B,O_2), C = \text{“}Paris\text{”}$$

Notice that Q is not supported by $\mathcal{P}$ in the absence of constraints (the source does not even allow views mentioning the bus predicate): $\textsc{Supp}^{\emptyset}_{\mathcal{P}}(Q)$ *does not hold.*

Assume that the source satisfies $\mathcal{C}$ which contains the inclusion dependency (4.13) below, stating that an operator will also cover by bus any leg important enough to be covered by train.

$$\forall X, Y, O \qquad t(X,Y,O) \longrightarrow b(X,Y,O) \tag{4.13}$$

Since $\mathcal{C}$ is (trivially) a weakly acyclic set of INDs, by Corollary 4.4.1 $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *holds if and only if so does* $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

Chase steps apply on the extensional parts of the second and third rules of $\mathcal{P}$, yielding the new rules (we underline the newly added tuples):

$$
\begin{aligned}
ind(B,C,O) \quad &:- \quad t(B,B',O), \underline{b(B,B',O)}, ind(B',C,O) \\
ind(B,C,O) \quad &:- \quad t(B,C,O), \underline{b(B,C,O)}
\end{aligned}
$$

The new program $chase_{\mathcal{C}}(\mathcal{P})$ *generates the views* V_{ij} *denoting the expansion with* i *legs from the origin to the intermediary point and* j *legs from the intermediary point to the destination. This includes the view* V_{11}*, which gives the shortest itineraries:*

$$(V_{11}) \quad v(A,B,C,O) :- t(A,B,O), b(A,B,O), t(B,C,O), b(B,C,O)$$

By chasing also the query, we obtain $Q' = chase_{\mathcal{C}}(Q)$*:*

$$
\begin{aligned}
(Q')\ q(A,B) \quad &:- \quad t(A,C,O_1), \underline{b(A,C,O_1)}, b(C,B,O_1), \\
&\qquad\quad t(C,B,O_2), \underline{b(C,B,O_2)}, C = \text{“}Paris\text{”}
\end{aligned}
$$

Observe that $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ *(and* $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$*) still does not hold because all the views* V_{ij} *require that only one operator be used. To enforce this requirement on* Q'*, one would need a constraint enforcing that the subgoals* $b(C,B,O_1)$ *and* $b(C,B,O_2)$ *from* Q' *refer to the same operator, making the equality* $O_1 = O_2$ *hold.*

Key safety. We next introduce the notion of a program being “key-safe”, which guarantees $\mathcal{C}$-locality and $\mathcal{C}$-independence in the presence of key constraints.

Let R be a relation with an n-attribute composite key and let $\bar{P} = (p_1, \ldots, p_k)$ be an ordered sequence of k distinct values in the range 1 to n. We say that a rule of $\mathcal{P}$ *outputs the key of* R*, by positions* $\bar{P}$*, into the sequence of head variables* $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$ if $\bar{X}$ appears in the rule body either

- in the positions $p_1, \ldots, p_k$ of the key attribute sequence of some R-subgoal, with the remaining $n-k$ positions (if any) of the key being bound to constant values, or
- in the positions $j_1, \ldots, j_k$ of some p-subgoal, where p is an IDB predicate with at least one rule that in turn outputs the key of R by key positions $\bar{P}$ into the sequence of head variables with indices $j_1, \ldots, j_k$.

We say that a subgoal g outputs the key of R, by positions $\bar{P} = (p_1, \ldots, p_k)$, into the sequence of variables $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$ if

- g uses EDB predicate R and $\bar{X}$ appears in positions $p_1, \ldots, p_k$ in the key attributes of g, with the remaining $n - k$ positions (if any) of the key being bound to constant values, or
- g uses IDB predicate p and there exists some rule defining p which outputs the key of R, by the key positions $\bar{P}$, into variables $\bar{X}$.

We say that a rule is *safe* for the key constraint on R if whenever one of its IDB subgoals outputs the key of R by some sequence of k key positions $\bar{P}$ into k variables $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$, no other subgoal does the same (for the same key positions $\bar{P}$). Notice that several EDB subgoals may output the key of the same R by the same key positions and into the same sequence of variables $\bar{X}$, as long as no IDB goal does.

EXAMPLE 4.4.2. *Suppose that, in Example 4.4.1, $\mathcal{C}$ contains also a key constraint on the b table, stating that bus operators cover disjoint legs:*

$$\forall X, Y, O \qquad b(X, Y, O), b(X, Y, O') \longrightarrow O = O' \tag{4.14}$$

Notice that $chase_{\mathcal{C}}(\mathcal{P})$ is the same as in Example 4.4.1 because no chase step applies with the key constraint.

The rules in $chase_{\mathcal{C}}(\mathcal{P})$ are safe. Indeed, in the second rule, b outputs the key into the sequence B, B', while ind outputs it into B', C. The two subgoals in the first rule also output the key, but into different sequences: A, B and B, C respectively.

Intuitively, safety of the rules in a program $\mathcal{P}$ is designed to guarantee $\mathcal{C}$-locality. It disallows two IDB goals in a rule from outputting the key of some EDB R into the same variables because this could lead, in the expansion of the rule, to two R goals agreeing on the key attributes and thus triggering a chase step with the key constraint. Since the R goals would come from the expansion of distinct IDB goals in the rule, the effect of this chase would not be reproducible by chasing the program rules in isolation (as in the definition of $chase_{\mathcal{C}}(P)$).

We now give a condition ensuring that every set of views generated by $\mathcal{P}$ is $\mathcal{C}$-independent. This requires additional restrictions on the rules of the distinguished predicates.

Definition 4.4.3. *A* program $\mathcal{P}$ is key-safe *for a set of key constraints $\mathcal{K}$ if*

1. *each rule is safe for all key constraints in $\mathcal{K}$, and*
2. *for all distinguished predicates* ans *of $\mathcal{P}$, all defining rules r of* ans*, and all relational symbols R in the schema, if r outputs the key attributes $\bar{A}$ (as defined above) of some goal $R(\bar{A}, \bar{B})$, it also outputs all non-key attributes $\bar{B}$ (by the same definition that applied to the key attributes).*

If $\mathcal{I}$ is a set of weakly acyclic INDs, we say that $\mathcal{P}$ is key-safe for $\mathcal{C} = \mathcal{K} \cup \mathcal{I}$ if $chase_{\mathcal{I}}(\mathcal{P})$ is key-safe for $\mathcal{K}$.

Note that key-safety can be checked in PTIME in the size of $\mathcal{P}$ and $\mathcal{K}$.

EXAMPLE 4.4.3. *Continuing Example 4.4.2, we observe that distinguished predicate ans outputs the pairs of key attributes A, B and B, C, but it also outputs O, the only non-key attribute. Therefore, $\mathcal{P}$ is key-safe.*

Intuitively, the key safety condition on the distinguished predicates ensures that, given query R' in terms of some views $\mathcal{V}$ generated by $\mathcal{P}$, there is query $R \equiv_{\mathcal{C}}^{\mathcal{V}} R'$ such that no chase step with a key constraint will apply to $expand_{\mathcal{V}}(R)$. This is because, if two view atoms in R' happen to output the key of some EDB goal G into the same variables $\bar{A}$, then by key-safety they each must also output all non-key attributes of G, say in variables $\bar{B}_1$, respectively $\bar{B}_2$. But then there is a query R, equivalent to R', obtained by adding to R' the equalities $\bar{B}_1 = \bar{B}_2$.

This equality is preserved in $expand_{\mathcal{V}}(R)$, so the chase step with the key constraint does not apply on $expand_{\mathcal{V}}(R)$. More formally, we can show the following.

Theorem 4.4.4. *Let $\mathcal{C}$ consist of key constraints and an acyclic set of inclusion dependencies. Any Datalog program $\mathcal{P}$ that is key-safe for $\mathcal{C}$ is also $\mathcal{C}$-local and all views generated by it are $\mathcal{C}$-independent.*

Proof: $\mathcal{C}$-locality follows from the fact that Claim 1 in the proof of Theorem 4.4.3 holds also for $\mathcal{C} = \mathcal{I} \cup \mathcal{K}$, where $\mathcal{I}$ is a set of INDs, and $\mathcal{K}$ a set of key constraints.

The proof is similar, using for the induction step a few additional observations about the chase with key constraints and INDs.

A first observation is that the chase of any query α with $\mathcal{C}$ yields a result that is equivalent to that obtained by first chasing with $\mathcal{I}$, then with $\mathcal{K}$:

$$chase_{\mathcal{C}}(\alpha) \equiv chase_{\mathcal{K}}(chase_{\mathcal{I}}(\alpha)),$$

as the key constraints from $\mathcal{K}$ never introduce new variables or new relational atoms.xo

A second observation is the following. Since $\mathcal{P}$ is key-safe for $\mathcal{C}$, this means by definition that $chase_{\mathcal{I}}(\mathcal{P})$ is key-safe for $\mathcal{K}$. This in turn implies that the result of $chase_{\mathcal{K}}(chase_{\mathcal{I}}(\alpha_i))$ can be alternatively obtained by expanding $chase_{\mathcal{K}}(chase_{\mathcal{I}}(\alpha_{i-1}))$ with $chase_{\mathcal{C}}(r_i)$. This is because the chase steps with INDs, as shown in the proof of Theorem 4.4.3, are triggered by single subgoals and not by the interaction of the EDB goals in α_{i-1} and the new goals in α_i. The same isolation property holds for the chase steps with key constraints. Key constraints (when expressed as dependencies) do have two goals in the premise, but due to the key safety restriction (see condition 1 in the definition of key-safety), the image of the premise can never span the EDB goals in α_{i-1} and the new EDB goals in α_i.

$\mathcal{C}$-independence follows similarly. Say that the set of generated views is $\mathcal{V}$, and CR is a canonical rewriting of query Q using $\mathcal{V}$.

The way in which views are joined in CR depends on Q, and therefore CR could conceivably contain two view subgoals both of which output the key of some

relation into the same variables $\bar{X}$. But then by construction of the canonical rewriting, Q itself joins two R-goals on the key, $R(\bar{X}, \bar{Y})$ and $R(\bar{X}, \bar{U})$. But then the key constraint applies when chasing Q (and this is a step in constructing the canonical rewriting), so $\bar{U}$ must be the same variables as $\bar{Y}$ in $chase_{\mathcal{C}}(Q)$, and therefore in CR. Since by key-safety of $\mathcal{P}$, all views that output keys must also output the non-key attributes, the equality of $\bar{Y}$ and $\bar{U}$ is guaranteed in the expansion of CR, and the chase step of CR with R's key constraint *does not apply*. In summary, we obtain that

$$\begin{aligned} chase_{\mathcal{C}}(expand_{\mathcal{V}}(CR)) &= \\ chase_{\mathcal{K}}(chase_{\mathcal{I}}(expand_{chase_{\mathcal{C}}(\mathcal{V})}(CR))) &= && (4.15)\\ chase_{\mathcal{I}}(expand_{chase_{\mathcal{C}}(\mathcal{V})}(CR)) &= && (4.16)\\ expand_{chase_{\mathcal{I}}(chase_{\mathcal{C}}(\mathcal{V}))}(CR) &= && (4.17)\\ expand_{chase_{\mathcal{C}}(\mathcal{V})}(CR). && && (4.18) \end{aligned}$$

Here, (4.17) follows from Claim 2 in the proof of Theorem 4.4.3, since any set of views, including $chase_{\mathcal{C}}(\mathcal{V})$ is $\mathcal{I}$-independent. (4.18) follows from the fact that $chase_{\mathcal{C}}(\mathcal{V})$ yields views on which no further chase step with any constraint in $\mathcal{C}$ applies, in particular with the constraints in $\mathcal{I} \subseteq \mathcal{C}$. ■

Corollary 4.4.2. *If $\mathcal{C}$ consists of key constraints and an acyclic set of INDs and $\mathcal{P}$ is key-safe for $\mathcal{C}$, then for any query Q,* $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Expr}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ *and*
$\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$.

EXAMPLE 4.4.4. *Continuing Example 4.4.3, a chase step with* (4.14) *applies on Q', introducing the equality atom $O_1 = O_2$. With this,* $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$ *holds, as witnessed by the rewriting*

$$q(A, B) \ :- \ V_{11}(A, \text{“}Paris\text{”}, B, O).$$

4.4.1 Refined Version of Key-Safety

The notion of key-safety presented above keeps only track of the positions bound to constants, ignoring the actual constant values that may appear in these

positions. As a consequence, it may fail to detect decidable instances of expressibility or support, where the constraints can still be ignored after the program and the query have been chased. We give in this section a refined definition for the key-safety restriction that addresses this problem, allowing us to solve strictly more cases by reduction to the dependency-free case. This refined notion of key-safety is implied by the one described above and detects strictly more decidable cases.

Let R be a relation with an n-attribute composite key. By a *template of constants* (in short, template) for the key of R we denote a sequence of values $T = (v_1, \ldots, v_n)$, where each v_i can be either a constant value or a special value denoted *blank*. By the variable positions of T we denote the ordered sequence of positions $\bar{P}_T = (p_1, \ldots, p_k)$ of T that are occupied by *blank*.

We say that a rule of $\mathcal{P}$ *outputs the key of R, by template T, into the sequence of head variables* $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$ if $\bar{X}$ appears in the rule body either

- in the positions $\bar{P}_T = (p_1, \ldots, p_k)$ of the key attribute sequence of some R-subgoal, with the remaining $n - k$ positions (if any) of the key being bound to the constant values given in T.
- in the positions $j_1, \ldots, j_k$ of some p-subgoal, where p is an IDB predicate with at least one rule that in turn outputs the key of R by the template T, into the sequence of head variables with indices $j_1, \ldots, j_k$.

We say that a subgoal g outputs the key of R, by template T, into the sequence of variables $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$ if

- g uses EDB predicate R and $\bar{X}$ appears in positions $\bar{P}_T = (p_1, \ldots, p_k)$ in the key attributes of g, with the remaining $n - k$ positions (if any) of the key being bound to the constant values given in T, or
- g uses IDB predicate p and there exists some rule defining p which outputs the key of R, by the template $\bar{T}$, into variables $\bar{X}$.

We say that a *rule is safe* for the key constraint on R if whenever one of its IDB subgoals outputs the key of R by some template of constants T into k variables $\bar{X} = (X_{i_1}, \ldots, X_{i_k})$, no other subgoal does the same (for the same template T).

Notice that several EDB subgoals may output the key of the same R by the same template and into the same sequence of variables $\bar{X}$, as long as no IDB goal does. A *program $\mathcal{P}$ is key-safe* for a set of key constraints $\mathcal{K}$ if

- each rule is safe for all key constraints in $\mathcal{K}$, and
- for all distinguished predicates *ans* of $\mathcal{P}$, all defining rules r of *ans*, and all relational symbols R in the schema, if r outputs the key attributes $\bar{A}$, by some template, of some goal $R(\bar{A}, \bar{B})$, it also outputs all non-key attributes $\bar{B}$, by some template (using the same definition that applied to the key attributes).

If $\mathcal{I}$ is a set of weakly acyclic INDs, we say that $\mathcal{P}$ is key-safe for $\mathcal{C} = \mathcal{K} \cup \mathcal{I}$ if $chase_{\mathcal{I}}(\mathcal{P})$ is key-safe for $\mathcal{K}$. Notice that this new definition of key-safety can still be checked in PTIME in the size of $\mathcal{P}$ and $\mathcal{K}$.

EXAMPLE 4.4.5. *Assuming the schema and constraints of Example 4.4.3, consider the following modified program $\mathcal{P}'$*

$$
\begin{array}{lrcl}
(\mathcal{P}') & ans(A,B,C,O) & :- & ind(A,B,O), ind'(B,O) \\
& ind(B,C,O) & :- & t(B,B',O), ind(B',C,O) \\
& ind(B,C,O) & :- & t(B,C,O) \\
& ind'(B,O) & :- & ind_P(B,O), ind_{SD}(B,O) \\
& ind_P(B,O) & :- & t(B, \text{“}Paris\text{”}, O) \\
& ind_{SD}(B,O) & :- & t(B, \text{“}SanDiego\text{”}, O)
\end{array}
$$

Notice that $\mathcal{P}'$ is not key-safe under the weaker restriction, since the rule defining ind$'$ is not safe. But we can easily see that the two constants appearing in the second attribute of the key cannot be equated during the chase, and $\mathcal{P}'$ is indeed key-safe under the refined definition.

More precisely, in the sixth rule, b outputs the key into the sequence of variables B, by the template

$$T_1 = (blank, \text{“}SanDiego\text{”}).$$

Similarly, in the fifth rule, b outputs the key into the sequence of variables B, by the template

$$T_2 = (blank, \text{“}Paris\text{”}).$$

Since T_1 and T_2 are different, the rule defining ind$'$ is safe. Then, in the first rule, the ind$'$ subgoal outputs the key in B, by any of these two templates. Finally, ans outputs the key attributes in A, B (by the ind subgoal) and in B (by the ind$'$ subgoal) but in both cases it also outputs O, the non-key attribute.

4.4.2 Inter-reducibility Preserves Decidability Restrictions

According to the results presented so far in this section, and in Section 4.3, under the decidability restrictions ($\mathcal{C}$-independence and $\mathcal{C}$-locality), we can solve expressibility under $\mathcal{C}$ even by using our favorite solver for dependency-free support (first reduce to dependency-free expressibility, then reduce to dependency-free support). Symmetrically, we can solve support under $\mathcal{C}$ using any solver for dependency-free expressibility. It turns out that the same cross-use of solvers can be achieved by first reducing from expressibility under $\mathcal{C}$ to support under $\mathcal{C}$ (using Theorem 4.3.2), and then to dependency-free support (using Theorem 4.4.1) (and symmetrically for support), as the reductions preserve restrictions for decidability.

Proposition 1. *Let Q be a conjunctive query, $\mathcal{C}$ a weakly acyclic set of dependencies, and $\mathcal{P}$ a $\mathcal{C}$-local Datalog program. Let Q', $\mathcal{C}'$ and $\mathcal{P}'$ be obtained, in PTIME, as in the reduction used in the proof of Theorem 4.3.1 such that we have* $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *iff* $\textsc{Expr}^{\mathcal{C}'}_{\mathcal{P}'}(Q')$. *Then (i) $\mathcal{P}'$ is $\mathcal{C}'$-local and (ii) if any finite set of views generated by $\mathcal{P}$ is $\mathcal{C}$-independent, then the views generated by $\mathcal{P}'$ are $\mathcal{C}'$-independent.*

Please also note that, as in Theorem 4.4.3, if $\mathcal{C}$ is a weakly acyclic set of inclusion dependencies, then so is $\mathcal{C}'$, hence $\mathcal{P}'$ is also $\mathcal{C}'$-local and the views it expresses are $\mathcal{C}'$-independent.

Proof: In the following, for a query Q, we will denote by $body_Q$ the conjunction of atoms in the body of Q.

(i) Any view V' generated by $\mathcal{P}'$ is of the form

$$\begin{aligned} ans'(Z_1, \ldots Z_m) :- \; & equals(Z_1, X_{i1}), \ldots, equals(Z_m, X_{im}), \\ & D(X_1), \ldots, D(X_m), \\ & body_{V_1}(Y_1^{(1)}, \ldots, Y_{k_1}^{(1)}, U_1^{(1)}, \ldots, U_{l_1}^{(1)}), \\ & equals(Y_1^{(1)}, X_{j1}^{(1)}), \ldots, equals(Y_{k_1}^{(1)}, X_{jk_1}^{(1)}), \\ & D(Y_1^{(1)}), \ldots, D(Y_{k_1}^{(1)}), D(U_1^{(1)}), \ldots, D(U_{l_1}^{(1)}), \\ & \ldots body_{V_n}(Y_1^{(n)}, \ldots, Y_{k_n}^{(n)}, U_1^{(n)}, \ldots, U_{l_n}^{(n)}), \\ & equals(Y_1^{(n)}, X_{j1}^{(n)}), \ldots, equals(Y_{k_n}^{(n)}, X_{jk_n}^{(n)}), \\ & D(Y_1^{(n)}), \ldots, D(Y_{k_n}^{(n)}), D(U_1^{(n)}), \ldots, D(U_{l_n}^{(n)}) \end{aligned}$$

where $V_i(Y_1^{(i)}, \ldots, Y_{k_i}^{(i)})$ are views generated by $\mathcal{P}$. Let us write it shortly

$$ans'(\bar{Z}) :- \; extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), body_{V_1}(\bar{Y}_1, \bar{U}_1), \ldots body_{V_n}(\bar{Y}_n, \bar{U}_n)$$

where $\bar{Y}$ and $\bar{U}$ are the union of all $\bar{Y}_i$ and all $\bar{U}_i$ variables, respectively. Since there are no D atoms in $\mathcal{C}'$, we have that $chase_{\mathcal{C}'}(V')$ is obtained by chasing only $body_{V_i}$, i.e.

$$\begin{aligned} & extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), chase_{\mathcal{C}'}(body_{V_1}(\bar{Y}_1, \bar{U}_1)), \ldots, \\ & chase_{\mathcal{C}'}(body_{V_n}(\bar{Y}_n, \bar{U}_n)) \end{aligned}$$

which is of the form

$$\begin{aligned} & extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), \\ & chase_{\mathcal{C}}(body_{V_1}(\bar{Y}_1, \bar{U}_1)), D(F_1^{(1)}), \ldots D(F_{p_1}^{(1)}), \ldots \\ & chase_{\mathcal{C}}(body_{V_n}(\bar{Y}_1, \bar{U}_1)), D(F_1^{(n)}), \ldots D(F_{p_n}^{(n)}) \end{aligned}$$

for some sets of variables $\bar{F}^{(i)} \subset \bar{U}_i \cup \bar{Y}_i$.

Since $\mathcal{P}$ is $\mathcal{C}$-local, for each V_i, there is a view W_i generated by $chase_{\mathcal{C}}(\mathcal{P})$ such that $chase_{\mathcal{C}}(V_i) \equiv W_i$. If we denote

$$w_i' = body_{W_i}(\bar{Y}_i, \bar{U}_i), D(F_1^{(i)}), \ldots D(F_{p_i}^{(i)})$$

we have that

$$chase_{\mathcal{C}'}(V') \equiv extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), w'_1(\bar{Y}_1, \bar{U}_1), \ldots$$
$$w'_n(\bar{Y}_n, \bar{U}_n).$$

Hence $chase_{\mathcal{C}'}(V')$ is equivalent to a view W' generated by $chase_{\mathcal{C}'}(\mathcal{P}')$ because chase steps only apply on the rules of $\mathcal{P}'$ obtained from the rules of $\mathcal{P}$ (by adding D atoms) and all the w'_i can be obtained by chasing the rules inherited from $\mathcal{P}$.

For the converse, we consider a view W' generated by $chase_{\mathcal{C}'}(\mathcal{P}')$. Using the same observation, W' has the subgoals from the bodies of n views $W_1, \ldots W_n$ generated by $chase_{\mathcal{C}}(\mathcal{P})$ (for some $n \geq 1$) plus the ones in the conjunction *extra* defined above and some other D atoms introduced by the chase. As $\mathcal{P}$ is $\mathcal{C}$-local, for each W_i there is a view V_i generated by $\mathcal{P}$ such that $chase_{\mathcal{C}}(V_i) \equiv W_i$. Reasoning in the same manner as above, we can put all the V_i views together and obtain a view V' generated by $chase_{\mathcal{C}'}(\mathcal{P}')$ such that $chase_{\mathcal{C}'}(V') \equiv W'$.

(ii) Consider n views $V'_1, \ldots, V'_n$ expressed by $\mathcal{P}'$. Each V'_i is of the form

$$V'_i(\bar{Y}_i) :- V_1^{(i)}(\bar{Y}_1^{(i)}), \ldots, V_{k_i}^{(i)}(\bar{Y}_{k_i}^{(i)}), extra_i(\bar{Z}_i, \bar{X}_i, \bar{Y}_i, \bar{U}_i)$$

where $\bigcup_{j=1}^{k_i} \bar{Y}_{k_i}^{(i)} = \bar{Y}_i$ and $extra_i$ is a conjunction of D and *equals* predicates, similar to *extra* from (i). Let *extra* be the conjunction of all the $extra_i$ and R be a query over $\{V'_1, \ldots, V'_n\}$. Please note that the chase with $\mathcal{C}'$ of $expand_{\{V'_1,\ldots,V'_n\}}(R)$ will only apply to the bodies of the $V_j^{(i)}$ views generated by $\mathcal{P}$. It follows that

$$chase_{\mathcal{C}'}(expand_{\{V'_1,\ldots,V'_n\}}(R))$$

is equivalent to a query having a body of the form

$$extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), chase_{\mathcal{C}'}\left(expand_{\{V_1,\ldots,V_n\}}(conj_{\mathcal{P}}(\bar{Y}, \bar{U}))\right),$$

where *extra* is a conjunction of $extra_i$ subqueries and $conj_{\mathcal{P}}$ is a conjunction of views V_i generated by $\mathcal{P}$.

Since $\mathcal{C}'$ is obtained from $\mathcal{C}$ by adding D atoms in the conclusions, the latter conjunction is equivalent to:

$$extra(\bar{Z}, \bar{X}, \bar{Y}, \bar{U}), chase_{\mathcal{C}}\left(expand_{\mathcal{V}}(conj_{\mathcal{P}}(\bar{Y}, \bar{U}))\right), \bigwedge D(F_{ijk})$$

where the $D(F_{ijk})$ subgoals are added by the conclusions of dependencies from $\mathcal{C}'$.

But we assumed the views generated by $\mathcal{P}$ to be $\mathcal{C}$-independent. Hence there is a query $T \equiv_{\mathcal{C}}^{\mathcal{V}} conj_{\mathcal{P}}$ (T outputs all the variables of $conj_{\mathcal{P}}$) such that

$$chase_{\mathcal{C}}(expand_{\{V_1,\ldots,V_n\}}(T)) \equiv$$
$$expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(T). \tag{4.19}$$

Let T' be the query obtained from T in the following way. We replace every view atom V_i (from $\mathcal{V}$) with a V_j', (from $\mathcal{V}'$) such that

- V_j' is constructed using only one expansion of the second rule (from program $\mathcal{P}'$) for the *temp* IDB predicate, such that the variables $\bar{X}$ of the *ans* predicate are the output variables of the $V_i(\bar{X})$ atom;
- notice also that the equalities between pairs of output variables of $V_j'(\bar{X})$ are already satisfied in T because those equalities are needed in order for $conj_{\mathcal{P}}$ to map into T.

From $T \equiv_{\mathcal{C}}^{\mathcal{V}} conj_{\mathcal{P}}$, it follows that

$$chase_{\mathcal{C}}(expand_{\mathcal{V}}(T)) \equiv chase_{\mathcal{C}}(expand_{\mathcal{V}}(conj_{\mathcal{P}})).$$

The mappings witnessing the latter equivalence can be extended to show that

$$chase_{\mathcal{C}'}(expand_{\mathcal{V}'}(T')) \equiv chase_{\mathcal{C}'}(expand_{\mathcal{V}'}(R)),$$

proving that $R \equiv_{\mathcal{C}'}^{\mathcal{V}'} T'$. We can extend the mapping because chasing with $\mathcal{C}'$ instead of $\mathcal{C}$ only brings D goals that can be inferred using bodies of views from $\mathcal{V}$ (there is no D goal in the premise of a rule from $\mathcal{C}'$). Hence $expand_{\mathcal{V}'}(T')$ and $expand_{\mathcal{V}'}(R)$ will behave the same way during the chase, since their subqueries based on $\mathcal{V}$ views, $expand_{\mathcal{V}}(T)$ and $conj_{\mathcal{P}}$ respectively, are equivalent. The rest of the subgoals are D atoms coming from the expansions of the views from $\mathcal{V}'$ (which are formed by bodies of views from $\mathcal{V}$ plus D atoms).

From equivalence (4.19) we can also infer that

$$chase_{\mathcal{C}'}(expand_{\{V_1',\ldots,V_n'\}}(T')) \equiv$$
$$expand_{\{chase_{\mathcal{C}'}(V_1'),\ldots,chase_{\mathcal{C}'}(V_n')\}}(T')$$

because the D atoms, including those from the bodies of $\mathcal{V}'$ views, are not involved in the chase and because chasing with $\mathcal{C}'$ is only different from chasing with $\mathcal{C}$ in that D atoms are added. But, by construction of $\mathcal{C}'$, the variables in these D atoms are variables that already existed in (4.19) hence the mappings witnessing (4.19) extend to the D atoms.

Since $R \equiv^{\mathcal{V}'}_{\mathcal{C}'} T'$, we can conclude that the views generated by $\mathcal{P}'$ are $\mathcal{C}'$-independent. ■

Proposition 2. *Let Q be a conjunctive query, $\mathcal{C}$ a weakly acyclic set of dependencies, and $\mathcal{P}$ a $\mathcal{C}$-local Datalog program. Let Q'', $\mathcal{C}''$ and $\mathcal{P}''$ be obtained, in PTIME, as in the reduction used in the proof of Theorem 4.3.2 such that we have $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ iff $\textsc{Supp}^{\mathcal{C}''}_{\mathcal{P}''}(Q'')$. Then (i) $\mathcal{P}''$ is $\mathcal{C}''$-local and (ii) if every finite set of views generated by $\mathcal{P}$ is $\mathcal{C}$-independent, then the views generated by $\mathcal{P}''$ are $\mathcal{C}''$-independent.*

Proof: Notice that the constraints do not mention *head* predicate used in the definition of Q', hence the *head* atoms are not involved in the chase.

Let V'' be a view generated by $\mathcal{P}''$ of the form

$$V''() :- \mathit{head}(\bar{X}), \mathit{body}_{\mathcal{P}''}(\bar{X}, \bar{Y}, U).$$

One can see that if we replace all EDB predicates of $\mathit{body}_{\mathcal{P}''}$ with predicates that do not use the U variable, we obtain the unfolding of a view V generated by $\mathcal{P}$, whose body is a conjunction $\mathit{body}_{\mathcal{P}}(\bar{X}, \bar{Y})$.

Under the assumption that $\mathcal{P}$ is $\mathcal{C}$-local, for every view V generated by $\mathcal{P}$, there is a view W generated by $\mathit{chase}_{\mathcal{C}}(\mathcal{P})$ such that $\mathit{chase}_{\mathcal{C}}(V) \equiv W$. We can prove by induction on the length of the chase sequence that there is a terminating chasing sequence for $\mathit{body}_{\mathcal{P}''}(\bar{X}, \bar{Y}, U)$ similar to the one for $\mathit{body}_{\mathcal{P}}(\bar{X}, \bar{Y})$ modulo the transformation of the EDB predicates. Hence there is also a view W'' produced by $\mathit{chase}_{\mathcal{C}''}(\mathcal{P}'')$ with $\mathit{chase}_{\mathcal{C}''}(V'') \equiv W''$.

To conclude (i), we can show that, conversely, for each view W'' generated by $\mathit{chase}_{\mathcal{C}''}(\mathcal{P}'')$ there is a corresponding W generated by $\mathit{chase}_{\mathcal{C}}(\mathcal{P})$ and a V generated by $\mathcal{P}$ such that $W \equiv \mathit{chase}_{\mathcal{C}}(V)$ implies $W'' \equiv \mathit{chase}_{\mathcal{C}''}(V'')$, where body of V'' is formed by body of V and a *head* atom. For that, we can use the same argument, namely the isomorphism between the two chase sequences.

To prove (ii), let R'' be a query formulated in terms of views $\mathcal{V}''$. Let R' be a query in terms of $\mathcal{V}$, obtained from R'' by removing the *head* predicates and the replacing the subgoals with corresponding subgoals on the original schema, by removing the U variable. Since the views produced by $\mathcal{P}$ are $\mathcal{C}$-independent, there is a query $R \equiv^{\mathcal{C}}_{\mathcal{V}} R'$ such that $chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)) \equiv expand_{\{chase_{\mathcal{C}}(V_1),\ldots\}}(R)$. Let then T be the query obtained from R by introducing back all the *head* atoms that were removed and by replacing the other subgoals with predicates that have one more variable, U. By extending the mappings that witness $R \equiv^{\mathcal{C}}_{\mathcal{V}} R'$ to predicates with the arity increased by one and to the *head* subgoals, we can also show that $T \equiv^{\mathcal{C}}_{\mathcal{V}} R''$. We conclude by noticing that $chase_{\mathcal{C}''}(expand_{\mathcal{V}''}(T)) \equiv expand_{\{chase_{\mathcal{C}''}(V_1''),\ldots\}}(T)$ follows from the similar property satisfied by R. ■

4.5 A Widely Applicable Sound Test

We next present a sound algorithm for testing support, applicable to any program and set of weakly acyclic dependencies. It is a decision procedure (no false negatives) under the decidability restrictions of Section 4.4, and in the dependency-free case (where it provides an exponentially better upper bound than previous work).

Our solution is based on the following overall strategy. Since a systematic enumeration of all (potentially infinitely many) views generated by a program $\mathcal{P}$ is infeasible, we instead "describe the behavior" (in a sense formalized shortly) of any view generated by $\mathcal{P}$ w.r.t. a decision procedure (described below) for the existence of a rewriting under $\mathcal{C}$ using *finitely* many views. This description will abstract away from the view body, focusing on how the view behaves in essential tests performed by this decision procedure. As it will turn out, under our decidability restrictions, there are only *finitely* many distinct behaviors, each exhibited by a possibly infinite set of views. It suffices therefore to find one representative view from each set, thus reducing the problem of checking support by $\mathcal{P}$ to checking the existence of a rewriting using the finitely many representatives. This problem is known to be decidable under weakly acyclic dependencies (Lemma 4.5.1 below).

We start by describing the associated decision procedure.

Canonical Rewriting Candidate. Given a finite set of views $\mathcal{V}$, an acyclic set of constraints $\mathcal{C}$, and a query Q, call the *canonical rewriting candidate* of Q using $\mathcal{V}$ under $\mathcal{C}$, denoted $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$, the query obtained as follows: (i) it has the same head variables as Q, and (ii) its body is constructed by evaluating each view $V \in \mathcal{V}$ over the body of $chase_{\mathcal{C}}(Q)$ (viewed as a symbolic database, also known as the canonical instance [AHV95]) and adding the subgoal $V(t)$ for every tuple t in the result of the evaluation.

We show next that the canonical rewriting candidate yields a decision procedure for the existence of a rewriting. This result reformulates a theorem in [DT03b] (see also [DPT06])[1]:

Lemma 4.5.1 (Corollary of [DT03b])**.** *Q has a rewriting using $\mathcal{V}$ under $\mathcal{C}$ iff $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$ is one. Moreover, this in turn holds iff (a) $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$ is safe (its head variables appear in its body), and (b) there is a containment mapping from Q into the result of chasing with $\mathcal{C}$ the expansion of $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$:*
$chase_{\mathcal{C}}(expand_{\mathcal{V}}(CRC_{\mathcal{V}}^{\mathcal{C}}(Q))) \sqsubseteq Q$.

EXAMPLE 4.5.1. *Revisiting Example 4.1.1, consider the following set of views $\mathcal{V} = \{V_1, V_2\}$:*

$$(V_1) \quad ans^1(Z_1, Z_2) \ :- \ f(Z_1, X), f(X, Z_2), t(Z_2, \text{“}Paris\text{”})$$
$$(V_2) \quad ans^2(Z_1, Z_2) \ :- \ f(Z_1, Y), f(Y, Z_2), b(Z_2, \text{“}Paris\text{”})$$

generated (among others) by $\mathcal{P}$. We will follow, step by step, the rewriting algorithm from [DT03b]. The first step consists in finding mappings from the view queries into the body of Q and adding, to Q, atoms corresponding to the head of the view query. V_1 is mapped into Q by $m_1 = \{Z_1 : A; X : C; Z_2 : B\}$, which leads to adding $ans^1(A, B)$. Similarly, for V_2 we discover the mapping

[1]Lemma 4.5.1 is a corollary of [DT03b], where it is also proven that there are only finitely many rewritings of Q using $\mathcal{V}$ that are minimal under $\mathcal{C}$, and that all of them are subqueries of $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$.

$m_2 = \{Z_1 : A; Y : C; Z_2 : B\}$ *and add* $ans^2(A, B)$. *We stop here, since no more mappings can be inferred. The result is an expanded query*

$$\begin{aligned} U: \quad q(A,B) :- \; & f(A,C), f(C,B), t(B, \text{“}Paris\text{”}), \\ & b(B, \text{“}Paris\text{”}), s(B, \text{“}Paris\text{”}), \\ & \underline{ans^1(A,B)}, \underline{ans^2(A,B)} \end{aligned}$$

in which the newly added atoms are underlined. U *is called the* universal plan *in [DT03b], and it is guaranteed that any exact rewriting of* Q *is a subquery of* U.

$R = CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ *is then obtained from* U *by keeping only the atoms from the view schema:*

$$R(A,B) :- ans^1(A,B), ans^2(A,B).$$

R *is equivalent to* Q *under dependency (4.1), as can be verified by first constructing the expansion* $E = expand_{\mathcal{V}}(CRC^{\mathcal{C}}_{\mathcal{V}}(Q))$ *as:*

$$\begin{aligned} E(A,B) \; :- \quad & f(A,X'), f(X',B), t(B, \text{“}Paris\text{”}), \\ & f(A,Y'), f(Y',B), b(B, \text{“}Paris\text{”}) \end{aligned}$$

which chases with (4.1) to query (cE):

$$\begin{aligned} cE(A,B) \; :- \quad & f(A,X'), f(X',B), t(B, \text{“}Paris\text{”}), \\ & f(A,Y'), f(Y',B), b(B, \text{“}Paris\text{”}), \\ & \underline{s(B, \text{“}Paris\text{”})} \end{aligned}$$

into which there is a containment mapping from Q, $cm_q = \{A : A, B : B, C : X'\}$. *The reverse containment also holds, as witnessed by the containment mapping from* cE *into* Q, $cm_e = \{A : A, B : B, X' : C, Y' : C\}$, *hence* R *is indeed a rewriting.*

Note that both views contribute to the rewriting, since both t *and* b *atoms are needed as images of the* t *and* b *atoms from* Q. *The contribution of* V_1 *consists in* m_{v1}, *a partial mapping of* Q *into* cE, *obtained by restricting the domain of* cm_q *to the first three atoms of* Q:

$$m_{v1} = \{A : A, B : B, C : X'\}.$$

In this case, the image of m_{v1}, E^1, is the entire expansion of ans^1:

$$E^1 = f(A, X'), f(X', B), t(B, \text{“}Paris\text{”}).$$

The contribution of V_2 is enabled by a partial mapping

$$m_{v2} = \{B : B\}$$

from (the b atom of) Q into the expansion of ans^2, with the image

$$E^2 = b(B, \text{“}Paris\text{”}).$$

m_{v1} and m_{v2} agree on the common B variable, and, since together they cover the whole of the body of Q, we obtain by combining them the containment mapping cm_q that maps the entire Q into cE.

Redundant views *Let us add now to program $\mathcal{P}$ a new rule, corresponding to the definition of the view V_3 given below:*

$$(V_3) \quad ans^3(Z_1, Z_3) :- f(Z_1, T), f(T, Z_2), b(Z_3, \text{“}Paris\text{”}).$$

Running the same rewriting algorithm as above on the set $\mathcal{V}' = \{V_1, V_2, V_3\}$, we discover that V_3 maps into Q by $m_3 = \{Z_1 : A, T : C, Z_2 : B, Z_3 : B\}$, which leads to a rewriting candidate $CRC^{\mathcal{C}}_{\mathcal{V}'}(Q)$ of the form

$$R'(A, B) :- ans^1(A, B), ans^2(A, B), ans^3(A, B).$$

V_3 does not modify the way in which the expansion query (which already had t and b atoms) chases, hence the resulting chased expansion of R' is:

$$\begin{aligned} cE'(A, B) \quad :- \quad & f(A, X'), f(X', B), t(B, \text{“}Paris\text{”}), \\ & f(A, Y'), f(Y', B), b(B, \text{“}Paris\text{”}), \\ & f(A, T'), f(T', T''), b(B, \text{“}Paris\text{”}), \\ & \underline{s(B, \text{“}Paris\text{”})} \end{aligned}$$

We can argue here that V_2 and V_3 are mutually redundant w.r.t. finding a rewriting of Q. The partial mapping $m_{v3} = \{B : B\}$ from Q into the expansion of ans^3, with

the image $b(B, \text{“}Paris\text{”})$, is isomorphic to the partial mapping m_{v2} from Q into the expansion of ans^2. To this, add the fact that both mappings from the bodies of the two views into Q, v_2 and v_3, agree on the images of the distinguished variables, mapping them into variables A and B of Q. Without going into further details, this would be enough to allow us to discard one of the two views and to obtain as a rewriting either $ans^1(A, B), ans^2(A, B)$ or $ans^1(A, B), ans^3(A, B)$.

According to Lemma 4.5.1 and the observations above, in order for a view to contribute to the rewritability of Q

(i) it must generate a subgoal g of the canonical rewriting candidate
e.g. V_1 generates $ans^1(A, B)$, introduced by the mapping m_1 from V_1 into Q;

(ii) g's expansion may participate in the chase with $\mathcal{C}$ of the expansion E of the canonical rewriting candidate
e.g. the expansion E^1 of $ans^1(A, B)$ contains the atom $t(B, \text{“}Paris\text{”})$, which, together with the expansion of V_2, $E^2 = b(B, \text{“}Paris\text{”})$, allows a chase step with dependency (4.1) to apply;

(iii) since Q maps into the chase of E, the expansion of g must include (after the chase) the image of a partial map from Q
e.g. E^1 is the image of m_{v1}.

We shall therefore describe a view V with respect to its behavior for (i), (ii) and (iii), using the notion of *descriptor*.

Normalized program. For uniformity of treatment, we will assume from now on w.l.o.g. that the program $\mathcal{P}$ is normalized as follows. For every k-ary IDB predicate p, every rule for p has the head variables $\bar{Z} = Z_1, \ldots, Z_k$, in that order. Furthermore, for every EDB predicate e, introduce a new IDB e', replace each occurrence of e in $\mathcal{P}$ with e', and add the rule $e'(\bar{Z}) :- e(\bar{Z})$. The normalized program has only two kinds of rules: those whose bodies consist of a single EDB subgoal (called *EDB rules*), or solely of IDB subgoals (called *IDB rules*). For technical reasons, we additionally compute (as in [LRU99]), the *closure* of the program, which consists in adding for every rule r in $\mathcal{P}$ all rules obtained from r by systematically equating in all possible ways the head variables of r with each other and with the constants in Q.

Definition 4.5.1 (Descriptors)**.** *For a query Q and a program $\mathcal{P}$, $E^{(p(t),fr)}$ is called a* descriptor *w.r.t Q and $\mathcal{P}$ iff*

- *p is an IDB predicate from $\mathcal{P}$,*
- *E is a conjunctive query body over EDBs from $\mathcal{P}$,*
- *$\mathcal{P}$ generates as expansion of p a query of head variables $\bar{Z}$, $p(\bar{Z}) :- body$,*
- *there is a homomorphism* $\mathrm{to} : body \rightarrow chase_{\mathcal{C}}(Q)$ *s.t.* $\mathrm{to}(\bar{Z}) = t$,
- fr *is a partial variable mapping from Q into $chase_{\mathcal{C}}(body)$ such that the image of Q under* fr *is E.*

We call E the expansion fragment *described by the descriptor, and $(p(t), fr)$ the* adornment *of E. We call variables $\{Z_1, \ldots, Z_k\}$ (where k is the arity of p) the* distinguished *variables of the descriptor, while all other variables in the range of fr are* hidden*.*

In the following, when referring to a descriptor we will omit the program $\mathcal{P}$ and the query Q if they are obvious from the context.

EXAMPLE 4.5.2. *In the setting of Example 4.5.1, $d_1 = E_1^{(p_1(t_1),fr_1)}$ and $d_2 = E_2^{(p_2(t_2),fr_2)}$ below are descriptors for the views V_1 and V_2, respectively:*

$$
\begin{array}{rcl}
d_1 & : & E_1 = [f(Z_1, X), f(X, Z_2), t(Z_2, \text{“}Paris\text{”})], \\
 & & p_1(t_1) = ans(A, B), fr_1 = \{A : Z_1, C : X, B : Z_2\} \\
d_2 & : & E_2 = [b(Z_2, \text{“}Paris\text{”})], \\
 & & p_2(t_2) = ans(A, B),\ fr_2 = \{B : Z_2\}
\end{array}
$$

Note that, though the two views contribute the same $ans(A, B)$ goal to the canonical rewriting candidate, the two descriptors distinguish among V_1 and V_2 by the images of Q into the view bodies (E_1 includes the image of Q's t and two f goals, E_2 only the b goal).

Before explaining in detail how descriptors are found, we show how they can be used to soundly infer support. Intuitively, a descriptor represents the fragment

of a chased view generated by $\mathcal{P}$ that serves as image of the partial mapping from Q. Our goal is to put together such fragments in a consistent way to create (if it exists) the image of Q under a *containment mapping*.

Partial rewriting candidate. More formally, consider a finite set of descriptors w.r.t. to query Q, program $\mathcal{P}$ and dependencies $\mathcal{C}$: $\mathcal{D} = \{E_i^{(p_i(t_i), fr_i)}\}_{1 \leq i \leq n}$, where all p_i are (not necessarily distinct) distinguished IDBs of $\mathcal{P}$. Introduce for each predicate p_i a fresh predicate p_i^i (using the rank i of the predicate in an arbitrary ordering of the descriptor set) such that $p_i^i \neq p_j^j$ for all $1 \leq i, j \leq n, i \neq j$. Assuming w.l.o.g. that Q's tuple of head variables is $\bar{X}$, we call the query

$$R(\bar{X}) :- p_1^1(t_1), \ldots, p_n^n(t_n)$$

the *partial rewriting candidate* described by $\mathcal{D}$. The set $\mathcal{V} := \{ VF_i : p_i^i(\bar{Z}) : - E_i\}_{1 \leq i \leq n}$ is called the *view fragments* described by $\mathcal{D}$. The view fragments VF_i are not necessarily safe queries, if not all the head variables serve as image of the partial mapping fr_i.

EXAMPLE 4.5.3. *For the set of descriptors $\mathcal{D} = \{d_1, d_2\}$ from Example 4.5.2, the fresh view goals are ans^1, ans^2 respectively. The partial rewriting candidate described by $\mathcal{D}$ is*

$$R(A, B) :- ans^1(A, B), ans^2(A, B)$$

(it happens to coincide with the canonical rewriting candidate shown in Example 4.5.1). The view fragments are

$$\begin{array}{lllll} (VF_1) & ans^1(Z_1, Z_2) & :- & f(Z_1, X), f(X, Z_2), t(Z_2, \text{“}Paris\text{”}) \\ (VF_2) & ans^2(Z_1, Z_2) & :- & b(Z_2, \text{“}Paris\text{”}). \end{array}$$

Notice how VF_1's, VF_2's bodies are isomorphic to fragments of the bodies of V_1, respectively V_2 from Example 4.5.1. Also, VF_2 is not safe as variable Z_1 does not appear in the body.

The following result allows us to test support, as in Lemma 4.5.1, but using descriptors instead of explicit views. The key idea is to use the partial rewriting candidate instead of the canonical rewriting candidate.

Corollary 4.5.1 (of Lemma 4.5.1)**.** *Let $\mathcal{D}$ be a finite set of descriptors w.r.t. query Q, program $\mathcal{P}$ and dependencies $\mathcal{C}$: $\mathcal{D} = \{E_i^{(p_i(t_i),fr_i)}\}_{1\leq i\leq n}$. Denote with*

- *R the partial rewriting candidate described by $\mathcal{D}$,*
- *$\mathcal{V}$ the view fragments described by $\mathcal{D}$,*
- *E the expansion $expand_{\mathcal{V}}(R)$.*

If (a) R is safe and (b) there exists a containment mapping cfr from Q into $chase_{\mathcal{C}}(E)$, then Q is supported by $\mathcal{P}$ under $\mathcal{C}$.

We say that any set $\mathcal{D}$ as in Corollary 4.5.1 *witnesses support*. Notice that conditions (a) and (b) in Corollary 4.5.1 reformulate the corresponding conditions from Lemma 4.5.1 in terms of descriptors.

EXAMPLE 4.5.4. *The set of descriptors $\mathcal{D}$ in Example 4.5.3 witnesses support for the query, program and dependency in our running Example 4.1.1. Indeed, if we apply the test of Corollary 4.5.1 to the partial rewriting candidate R and the view fragments VF_1 and VF_2 described by $\mathcal{D}$(shown in Example 4.5.3), we obtain*

- *the expansion*

$$EF(A,B) \quad :- \quad f(A,X'), f(X',B), t(B, \text{“Paris”}), \\ b(B, \text{“Paris”})$$

- *the result (cEF) of chasing EF with dependency (4.1),*

$$cEF(A,B) \quad :- \quad f(A,X'), f(X',B), t(B, \text{“Paris”}), \\ b(B, \text{“Paris”}), \underline{s(B, \text{“Paris”})}$$

Notice that EF and cEF are fragments of E, respectively cE from Example 4.5.1. Let cfr be the mapping $\{A:A, B:B, C:X'\}$. Observe that (a) R is safe; and (b) cfr is a containment mapping from Q into cEF, thus satisfying the conditions of Corollary 4.5.1.

The number of descriptors is infinite due to the unbounded set of hidden variables, but there are only finitely many isomorphism types of descriptors modulo renaming of the hidden variables, in the following sense:

Definition 4.5.2 (Similarity)**.** *Two descriptors* $E_1^{(p_1(t_1),fr_1)}$ *and* $E_2^{(p_2(t_2),fr_2)}$ *are* similar *iff* $p_1 = p_2$ *(and hence the distinguished variables of the descriptors are the same),* $t_1 = t_2$*, and there is an isomorphism* i *between the ranges of* fr_1 *and* fr_2 *which is the identity on the distinguished variables, and* i *witnesses the isomorphism of* E_1 *and* E_2*.*

Intuitively, the condition on fr_1 and fr_2 ensures that the partial containment mapping of Corollary 4.5.1, restricted to the view fragment, is the same for both descriptors. It is easy to see that similarity is an equivalence relation, and that there are only finitely many equivalence classes of descriptors under similarity. Indeed in $E^{(p(t),fr)}$, p is a predicate from $\mathcal{P}$; t a tuple of variables and constants from $chase_{\mathcal{C}}(Q)$, thus the number of distinct values it can take is polynomial in the size of $chase_{\mathcal{C}}(Q)$ and exponential in the arity of p; the number of distinct (up to isomorphism) partial mappings fr is exponential in the number of variables in Q.

Similarity plays a key role in our support test. Indeed we can show that any representative of a similarity equivalence class is as good as any member of the class for the purpose of witnessing support, in the following sense:

(†) if descriptor d_1 is similar to d_2, then for any set $\mathcal{D}$ of descriptors, $\mathcal{D} \cup \{d_1\}$ is a support witness if and only if $\mathcal{D} \cup \{d_2\}$ is one.

Algorithm findDescriptors. We next present a bottom-up algorithm for computing representatives of descriptor equivalence classes under similarity. The algorithm **findDescriptors** consists in initializing a set of descriptors $\mathcal{D}$ to the empty set, then repeatedly carrying out the rule steps described below until $\mathcal{D}$ reaches a fixpoint (under similarity), finally returning $\mathcal{D}$.

EDB rule step. Consider an EDB rule

$$e'(Z_1, \ldots, Z_k) :- e(Z_1, \ldots, Z_k)$$

For every variable mapping *to* from $Z_1, \ldots, Z_k$ into Q's variables and constants, such that the goal $e(to(Z_1), \ldots, to(Z_k))$ appears in $chase_{\mathcal{C}}(Q)$; and every partial variable mapping *fr* from the variables of Q to $\{Z_1, \ldots, Z_k\}$ (including the empty-domain one), add to $\mathcal{D}$ the descriptor $E^{(e(to(\bar{Z})),fr)}$, where $E = e(\bar{Z})$. Note that descriptors with empty-domain mappings capture the situation when none of the query goals maps into the described e goal[2].

IDB rule step. Consider an IDB rule

$$p(\bar{X}) :- p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$$

If there exists a homomorphism h from the rule body into $chase_{\mathcal{C}}(Q)$, and a set of descriptors

$$E_1^{(p_1(h(\bar{X}_1)),fr_1)}, \ldots, E_n^{(p_n(h(\bar{X}_n)),fr_n)}$$

in $\mathcal{D}$, then:

Construct the views $V_i : p_i(\bar{Z}_i) :- E_i$. Denote with E the expansion of the rule body using these views, and with xh_i the corresponding expansion homomorphism $xh_i : E_i \to E$ (i.e. the variable renaming performed on each V_i during expansion). Chase E with $\mathcal{C}$ and denote with ch the corresponding chase homomorphism $ch : E \to chase_{\mathcal{C}}(E)$. If the set $\{ch \circ xh_i \circ fr_i\}_{1 \le i \le n}$ of partial mappings from Q into $chase_{\mathcal{C}}(E)$ is compatible, construct the combined mapping $cfr := \bigcup_{i=1}^{n} ch \circ xh_i \circ fr_i$, otherwise exit the rule step. For every partial mapping *fr* from Q into $chase_{\mathcal{C}}(E)$ which extends *cfr* (including the trivial extension $fr = cfr$) by mapping additional variables of Q into fresh variables added during the chase, compute the descriptor $d = F^{(p(h(\bar{X})),fr)}$, where F is the image under *fr* of all goals in Q such that *fr* is defined on all their variables. If d is not similar to any descriptor in $\mathcal{D}$, add it to $\mathcal{D}$.

[2]Technically, descriptors for EDB rule IDBs using empty-domain partial mappings do not fully conform to Definition 4.5.1 as the expansion fragment contains a goal that is not the image under the partial mapping. As seen in the IDB rule step, the definition holds for all other IDBs, which are the pre-normalization IDBs.

EXAMPLE 4.5.5. *We next illustrate the rule steps of algorithm* **findDescriptors** *for Example 4.1.1 showing how descriptors d_1 and d_2 from Example 4.5.2 are derived. First, observe that no chase step applies on Q, so $Q = chase_{\mathcal{C}}(Q)$.*

For brevity, we work on the unnormalized program $\mathcal{P}$. Applications of EDB rule steps produce (among others) the following descriptors:

$$d_3 = [f(Z_1, Z_2)]^{(f(A,C),\{A:Z_1,C:Z_2\})}$$
$$d_4 = [f(Z_1, Z_2)]^{(f(A,C),\{\})}$$
$$d_5 = [f(Z_1, Z_2)]^{(f(C,B),\{C:Z_1,B:Z_2\})}$$
$$d_6 = [f(Z_1, Z_2)]^{(f(C,B),\{\})}$$
$$d_7 = [t(Z_1, \text{“}Paris\text{”})]^{(t(B,\text{“}Paris\text{”}),\{B:Z_1\})}$$
$$d_8 = [b(Z_1, \text{“}Paris\text{”})]^{(b(B,\text{“}Paris\text{”}),\{B:Z_1\})}.$$

Notice that for the same match of EDB goal $f(Z_1, Z_2)$ into goal $f(A, B)$ of $chase_{\mathcal{C}}(Q)$, several partial mappings from the query are considered. We show only two here (in descriptors d_3 and d_4, where the latter uses the empty mapping, meaning that no query variable is mapped into its fragment).

An IDB rule step for the fourth $\mathcal{P}$ rule combines the descriptors d_5 and d_7 yielding a new descriptor:

$$(d_9) \quad [f(Z_1, Z_2), t(Z_2, \text{“}Paris\text{”})]^{(ind(C,B),\{C:Z_1,B:Z_2\})}$$

which combines with d_3 using the first rule of $\mathcal{P}$, yielding d_1.

Descriptors d_6 and d_8 combine via an IDB rule step with the third rule in $\mathcal{P}$ to

$$(d_{10}) \quad [b(Z_2, \text{“}Paris\text{”})]^{(ind(C,B),\{B:Z_2\})}$$

which combines with d_4 using the first rule of $\mathcal{P}$, yielding d_2.

We next prove that the inflationary process for descriptor discovery implemented by algorithm **findDescriptors** always terminates for weakly acyclic sets of constraints.

Lemma 4.5.2. *If $\mathcal{C}$ is weakly acyclic, then algorithm* **findDescriptors** *is guaranteed to*

(a) terminate in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$, and Q.

(b) output only descriptors, which are all pairwise dissimilar.

Proof: (a) Notice that the initialization stage and each individual rule step terminate, since the chase terminates when $\mathcal{C}$ is weakly acyclic. The set $\mathcal{D}$ must saturate, as there are only finitely many dissimilar descriptors. Their number is upper bounded by an exponential in the maximum arity of a predicate in $\mathcal{P}$ and the size of Q, which bounds the number of rule step applications. At every rule step, finding that the rule applies involves matching it against the set of descriptors, which is exponential in the rule size. By Proposition 1 from [DT03b], the ensuing chase terminates in time exponential in the size of $\mathcal{C}$ and polynomial in the size of the descriptor.

(b) An easy proof by induction on the structure of the derivation tree of each descriptor. ■

Algorithm testSupport. Our algorithm for testing support amounts to deciding if the descriptors computed by algorithm **findDescriptors** give a support witness (in the sense of Corollary 4.5.1). According to Corollary 4.5.1, the existence of such a witness is sufficient for support, but, due to our undecidability results, when the program is unrestricted (see Section 4.6), it is not always a necessary condition. That is why algorithm *testSupport* is in general only sound.

algorithm testSupport
input: query Q, program $\mathcal{P}$, set of dependencies $\mathcal{C}$;

begin
$\mathcal{N}$:= the normalization of $\mathcal{P}$;
$\mathcal{D}$:= **findDescriptors**$(Q, \mathcal{N}, \mathcal{C})$;
$\mathcal{D}'$:= all descriptors from $\mathcal{D}$ pertaining to
distinguished predicates of $\mathcal{N}$;
if $\mathcal{D}'$ witnesses support (tested as in in Corollary 4.5.1)
then return true;
else return false;

end

Algorithm **testSupport** satisfies the following properties.

Theorem 4.5.1. *If $\mathcal{C}$ is weakly acyclic, the following hold: (1) algorithm* **testSupport** *is sound for testing support, and (2) it runs in time exponential in the size of $\mathcal{P}$, $\mathcal{C}$, and Q.*

Proof: (1.) follows from Lemma 4.5.2(b) and Corollary 4.5.1.

(2.) follows from Lemma 4.5.2(a) and Corollary 4.5.1, noticing that the containment mapping *cfr* can be computed in EXPTIME in the size of Q and in PTIME in the size of the result of chasing the partial rewriting candidate. In turn, the size of the chase result is exponential in the maximum arity of a constraint in $\mathcal{C}$ and polynomial in that of the partial rewriting candidate [DT03b]. The size of the partial rewriting candidate is given by the maximum number of distinct descriptors that can be built, which by Lemma 4.5.2(a) is worst-case exponential in the size of Q, $\mathcal{C}$, and the maximum arity of a predicate in $\mathcal{N}$(which remains unchanged during normalization, so it coincides with the maximum arity of a predicate in $\mathcal{P}$). Notice from the proof of Lemma 4.5.2 that only the maximum arity a of the program $\mathcal{N}$, and the size s of its rules may appear in the exponent. The number of rules in $\mathcal{N}$ does not appear in the exponent. It is easy to check that the normalization process affects only the number of rules (which blows it up exponentially from $\mathcal{P}$ to $\mathcal{N}$) and preserves the values a and s from $\mathcal{P}$. ■

Algorithm **testSupport** produces strictly less false negatives than the approach of reducing away dependencies described in Section 4.4. First, it is a decision procedure whenever the reduction succeeds:

Theorem 4.5.2. *If $\mathcal{C}$ is weakly acyclic and $\mathcal{P}$ is a $\mathcal{C}$-local program generating $\mathcal{C}$-independent views, then algorithm* **testSupport** *is a decision procedure for support.*

Proof: We will show that, under the assumptions of the theorem, there is a rewriting of a query Q in terms of the views produced by $\mathcal{P}$ iff **testSupport** returns true. The *if* direction was proven as part of Theorem 4.5.1. We still

need to prove the *only if* part, i.e. for $\mathcal{P}$-local programs generating $\mathcal{C}$-independent views, the algorithm is also complete.

Suppose there is a rewriting RW of Q: $Q \equiv_{\mathcal{C}} expand_{\mathcal{V}}(RW)$. Since the views are $\mathcal{C}$-independent, there is a query R over $\mathcal{V}$ such that $RW \equiv_{\mathcal{C}}^{\mathcal{V}} R$ and

$$chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)) \equiv expand_{\{chase_{\mathcal{C}}(V_1),\ldots,chase_{\mathcal{C}}(V_n)\}}(R)$$

By $\mathcal{C}$-locality, it follows that there are views $\mathcal{W} = \{W_1, \ldots W_n\}$ generated by $chase_{\mathcal{C}}(\mathcal{P})$ such that $chase_{\mathcal{C}}(V_i) \equiv W_i$. Let T be the query obtained from R by substituting every V_i with the corresponding W_i. Then:

$$expand_{\mathcal{W}}(T) \equiv chase_{\mathcal{C}}(expand_{\mathcal{V}}(R)) \equiv chase_{\mathcal{C}}(Q) \tag{4.20}$$

where the last equivalence follows from $RW \equiv_{\mathcal{C}}^{\mathcal{V}} R$.

This means in particular there is a containment mapping m_1 from Q into $expand_{\mathcal{W}}(T)$ formed by combining partial mappings m_1^i from Q into the expansion of each view W_i. From (4.20) we also know there is a containment mapping m_2 from $expand_{\mathcal{W}}(T)$ into $chase_{\mathcal{C}}(expand_{\mathcal{V}}(R))$, hence $m_3 = m_2 \circ m_1 = \bigcup_i m_2 \circ m_1^i$ is also a containment mapping from Q into $chase_{\mathcal{C}}(expand_{\mathcal{V}}(R))$.

By running algorithm **testSupport** for query Q and program $chase_{\mathcal{C}}(\mathcal{P})$, we obtain a proof of $\textsc{Supp}^{\emptyset}_{chase_{\mathcal{C}}(\mathcal{P})}(chase_{\mathcal{C}}(Q))$. This follows from the existence of the containment mapping m_1 (which is a mapping in the absence of constraints) and can be proven by induction on the structure of the views. For each view subgoal W_i of T, the algorithm will compute a descriptor for which the partial mapping from Q is m_1^i. By combining them, we obtain m_1 as a containment mapping from Q into the expansion of T using view fragments from $\mathcal{W}$. The way T was built out of R and the $\mathcal{C}$-independence of $\mathcal{V}$ guarantee that when applying **testSupport** this time for $\mathcal{P}$(instead of $chase_{\mathcal{C}}(\mathcal{P})$) we obtain all possible descriptors corresponding to the subgoals of R (because the chase only applies locally to each view body and not across views). And (4.20) guarantees there is a containment mapping from Q into $chase_{\mathcal{C}}(expand_{\mathcal{V}}(R))$, hence also into the chased expansion that uses only view fragments. This implies that **testSupport** returns true also when called for program $\mathcal{P}$ and the set of constraints $\mathcal{C}$. ■

Corollary 4.5.2. *If $\mathcal{C}$ is a weakly acyclic set of key and foreign key constraints, and $chase_{\mathcal{C}}(P)$ is safe for the keys in $\mathcal{C}$, then* **testSupport** *is a decision procedure for support.*

Second, the setting of Example 4.1.1 exhibits a case in which the restrictions required in Section 4.4 for reduction to the dependency-free case do not apply (they involve keys and foreign keys, while dependency (4.1) is neither). Indeed, it is easy to check that the chased program does not support the chased query in the absence of dependencies. We therefore need a qualitatively better approach, which is provided by algorithm **testSupport**: Example 4.5.5 shows that the call to **findDescriptors** yields (among others) the descriptors d_1, d_2, which, according to Example 4.5.4, witness support.

Algorithm testExpressibility. While we could use the reduction from expressibility to support used in Theorem 4.3.2, the following variation on **testSupport** constitutes a direct test: call **findDescriptors**, keep only the descriptors for distinguished IDB predicates, and perform the test of Corollary 4.5.1 only on singleton sets of descriptors. The soundness of this algorithm follows directly from the soundness of **testSupport**.

Finding the actual views. So far, we have only provided algorithms for deciding support and expressibility. To turn them into algorithms exhibiting the actual views generated by $\mathcal{P}$ as well as the rewriting using it requires extra bookkeeping. All we need to do is to carry along with a descriptor d the actual expansion built during its derivation, noticing that the derivation tree of d coincides with the expansion tree of the expansion described by d.

Finding minimized witnesses for support. Let us note that while the partial rewriting candidate described by $\mathcal{D}'$ in algorithm **testSupport** may contain redundant atoms, in security applications we only need to check if a query is supported by authorized views [RMSR04], which amounts to checking the existence of a rewriting without ever using it. Instead, the original query is executed once it is authorized. For non-security applications in which the wrapper needs to find and execute the rewriting in order to service a user application, one can plug in any technique for minimization under constraints already studied in the literature. One

of them is the backchase minimization [DPT06] which starts from the rewriting candidate (corresponding to R from Corollary 4.5.1) and considers subsets of view atoms at a time. This technique is amenable to further optimization by reusing the information from the partial mappings stored in the descriptors: find subsets of descriptors whose partial mappings are compatible and yield a total mapping from the query into the partial rewriting candidate. The presentation of such an optimization algorithm combining the discovered descriptors more efficiently goes beyond the scope of this dissertation.

4.5.1 Revisiting the Dependency-free Case

Based on algorithm **testSupport**, we now improve the previously best-known upper bound for checking support in the dependency-free setting. [LRU99] reported a deterministic doubly-exponential upper bound in the size of the query and program. We obtain an exponentially improved upper bound, implied by Theorem 4.5.2 and Theorem 4.5.1:

Corollary 4.5.3. *In the absence of dependencies, algorithm* **testSupport**

(a) is a decision procedure for support of a query Q by an arbitrary program $\mathcal{P}$, and

(b) runs in deterministic EXPTIME in the sizes of $\mathcal{P}$ and Q.

We next show that this upper bound is tight in the program size, and tight for practical purposes in the query size.

Theorem 4.5.3. $\textsc{Supp}^{\emptyset}_{\mathcal{P}}(Q)$ *is NP-hard in the size of Q and EXPTIME-complete in the size of $\mathcal{P}$.*

Proof: The NP lower bound follows from a reduction from the problem of checking conjunctive query equivalence (NP-complete by [CM77]), via the problem of checking expressibility. Given conjunctive queries Q_1, Q_2, we have that $Q_1 \equiv Q_2$ iff Q_1 is expressible by the single-rule Datalog program Q_2. The latter reduces in PTIME to the problem of support by Theorem 4.3.2.

The EXPTIME lower bound is obtained by a reduction from the problem of checking containment of a query Q in a Datalog program $\mathcal{P}$, known to be PTIME

in the size of Q and EXPTIME-complete in the size of $\mathcal{P}$ [RSUV89]. First, we carry out a reduction to the problem of checking expressibility, then compose it with the PTIME reduction from expressibility to support given by Theorem 4.3.2:

Given query $Q(\bar{X}) :- \ body(\bar{X}, \bar{Y})$ and program $\mathcal{P}$ of distinguished predicate *ans* (necessarily of the same arity as the query), we construct program $\mathcal{P}'$ which includes all rules of $\mathcal{P}$, the additional rule $ans'(\bar{Z}) :- \ ans(\bar{Z}), body(\bar{Z}, \bar{Y})$ and pick ans' as the new distinguished predicate of $\mathcal{P}'$. Notice that $\mathcal{P}'$ generates all intersections of Q with views generated by $\mathcal{P}$, whence we have that Q is contained in $\mathcal{P}$ iff $\textsc{Expr}^{\emptyset}_{\mathcal{P}'}(Q)$ holds. ■

4.6 Boundaries of Decidability

We next justify the restrictions of Section 4.4 by exploring the boundaries of decidability for the problems of expressibility and support. To calibrate our results, we start with the following: allowing unrestricted sets of constraints immediately leads to undecidability even if the program expresses a single view. This result is unsurprising given that unrestricted sets of embedded dependencies notoriously lead to undecidability of many fundamental database decision problems, such as equivalence of queries and implication of dependencies [AHV95]:

Theorem 4.6.1. *If $\mathcal{C}$ contains arbitrary embedded dependencies,* $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *and* $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *are undecidable even if $\mathcal{P}$ expresses a single view.*

Proof: The undecidability of support follows from the undecidability of expressibility and the reduction of Theorem 4.3.2. As for the undecidability of expressibility, it follows from a reduction from query containment under embedded dependencies (known to be undecidable [AHV95]) to support of a query by a nonrecursive Datalog program which expresses a single view. ■

Theorem 4.6.1 shows that decidability requires the set of constraints to conform at least to the restrictions yielding decidability in the single-view case. The most permissive restriction known to date requires $\mathcal{C}$ to be a weakly acyclic set of embedded dependencies [DT03b, FKMP03]. As we show below, weak acyclicity turns out to be too generous for sets of views described by unrestricted programs.

Indeed, it turns out that the interaction of recursion in the program and the presence of dependencies leads to undecidability even under strong restrictions on the dependencies and on the program which are known to lead to decidability in many classical decision problems as long as recursion and dependencies are mutually exclusive. For instance, query rewritability using finitely many views (listed explicitly, not described by a program) is known to be decidable under weakly acyclic dependencies [DT03b], in particular under only functional dependencies (which include key constraints), or only full TGDs. In the absence of dependencies, expressibility and support for arbitrary recursive programs is decidable [LRU99]. Moreover, many classical undecidable Datalog-related problems, such as containment and boundedness (undecidable by [GMSV93]) are known to become decidable for recursive *monadic* programs [CGKV88]. However when considering recursion and dependencies together, we obtain surprisingly strong undecidability results.

Recall that a program is *monadic* if all its IDB predicates have arity 1, and it is *linear* if each rule body contains at most one intentional subgoal.

Theorem 4.6.2. *If $\mathcal{P}$ is recursive and not key-safe, then* $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ consists of a single key constraint, and $\mathcal{P}$ is a linear monadic program.*

Proof: The proof is by reduction from the Post Correspondence Problem (PCP), known to be undecidable [UH79, AHV95]. Let $\{v_i\}_{1\leq i\leq n}$, $\{w_i\}_{1\leq i\leq n}$ be the PCP instance, where v_i, w_i are words over alphabet $\{a, b\}$. This is a "yes" instance iff there exists a natural number l and a sequence of integers $\sigma \in \{1, \ldots, n\}^l$ such that

$$v_{\sigma(1)} \circ v_{\sigma(2)} \circ \ldots \circ v_{\sigma(l)} = w_{\sigma(1)} \circ w_{\sigma(2)} \circ \ldots \circ w_{\sigma(l)}$$

where $\sigma(i)$ denotes the i^{th} integer in the sequence, and $\circ$ is the word concatenation operator. Any such σ is called a *solution* of the PCP problem. Any sequence σ (regardless of whether it is a solution) determines a word obtained by concatenating the corresponding w-words, and one obtained by concatenating the corresponding v-words.

We construct a monadic, linear (recursive) Datalog program $\mathcal{P}$, the singleton set $\mathcal{C}$ comprising a key constraint, and a query Q such that the PCP problem

has a solution iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

We use only one EDB relation $e(X, l, Y)$, intended to denote a directed edge with source X, target Y and label l. The (boolean) query Q is the following, where all lower-case letters (e.g. l, r, a, b, c, d) are constants, and upper-case letters are variables:

$$\begin{aligned} Q() \ :- \quad & e(A, l, B), e(A, r, C), e(D, c, A), \\ & e(D, a, D), e(D, b, D), e(D, d, D). \end{aligned}$$

The program $\mathcal{P}$ is constructed as follows (again, lower-case letters are constants and upper-case letters are variables). $\mathcal{P}$ consists of

- the rule $V() :- \ C(X)$;
- the rule

$$\begin{aligned} C_r(X) \ :- \quad & e(X, d, Y), \\ & e(X, c, X'), e(X', l, Z), \\ & e(Y, c, Y'), e(Y', r, T), \\ & e(U, a, U), e(U, b, U), e(U, d, U), \\ & e(U, c, X'); \end{aligned}$$

- for every $1 \leq i \leq n$, assuming w.l.o.g. that $v_i = \alpha^i_1 \dots \alpha^i_{k_i}$ and $w_i = \beta^i_1 \dots \beta^i_{l_i}$, the rules

$$\begin{aligned} C(X) \ :- \quad & e(X, \alpha^i_1, X_1), \dots, e(X_{k_i - 1}, \alpha^i_{k_i}, X_{k_i}), \\ & e(X, \beta^i_1, Y_1), \dots, e(Y_{l_i - 1}, \beta^i_{l_i}, Y_{l_i}), \\ & e(X_{k_i}, d, Y_{l_i}), C_r(X_{k_i}); \\ C_r(X) \ :- \quad & e(X, d, Y), \\ & e(X, \alpha^i_1, X_1), \dots, e(X_{k_i - 1}, \alpha^i_{k_i}, X_{k_i}), \\ & e(Y, \beta^i_1, Y_1), \dots, e(Y_{l_i - 1}, \beta^i_{l_i}, Y_{l_i}), \\ & e(X_{k_i}, d, Y_{l_i}), C_r(X_{k_i}); \end{aligned}$$

$\mathcal{C}$ comprises just one key constraint, stating that the source and label of an edge determine its target:

$$\forall X, L, Y, Y' \; e(X, L, Y) \wedge e(X, L, Y') \rightarrow Y = Y'.$$

$\mathcal{P}$ is designed to generate, for every sequence σ of integers from $\{1, \ldots, n\}$, an expansion which encodes the two concatenations of v-words and w-words determined by σ. A word is encoded by a chain of edges, each edge label encoding a character in the word. The expansion thus contains two chains of words (one for the v_i's, one for the w_i's), each of them ended by a `c`-labeled edge followed by an `l`-edge, respectively an `r`-edge. The chains start from the same node (according to the C rule), and continue in parallel, chaining together pairs of subchains which correspond to pairs of words (v_i, w_i) for some i (this is the role of the repeated expansions of IDB C_r according to the rule for i). The expansion is ended by a subgraph given by the expansion of the first rule of C_r, whose role will be explained shortly.

To enable mappings from the arbitrarily long chains of the expansions into the query, Q contains cycles into which every pair of chains can map. Indeed, it is easy to see that any expansion of $\mathcal{P}$ has a containment mapping into Q. Since the cycles in Q cannot map into the straight chains in the expansions of $\mathcal{P}$, the v-chain is ended by the cycles generated by the first rule of C_r.

We therefore have that $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ holds iff $\mathcal{P}$ expresses some view V such that $V \sqsubseteq_{\mathcal{C}} Q$ (since the opposite containment holds for every expansion, even in the absence of constraints). Because $\mathcal{C}$ contains only a key constraint, the chase with it is guaranteed to terminate, and $V \sqsubseteq_{\mathcal{C}} Q$ holds iff $chase_{\mathcal{C}}(V) \sqsubseteq Q$ [AHV95].

Observe that successive expansions of the C_r IDB chain only the v-words together; the v_i-words in the expansion of each rule start from variable X which is also the end of the previous v_j-word in the concatenation, but the w_i-words start from the fresh variable Y which is not connected to the end Y_{k_j} of the previous w_j-word. Connecting the successive w-words explicitly would require IDB C_r to be binary, carrying both ends of v and w-words. To use only monadic rules, we rely instead on the key constraint: the variable beginning any w-word and the variable

ending the previous w-word in the chain are both targets of `d`-edges emanating from the junction of the previous and current v-word. The chase with the key constraint will "glue" the two chain segments corresponding to the w-words.

The intuition behind the construction is that, if we log for each one-step expansion of IDB C_r the i corresponding to the rule used, the obtained sequence of integers is the candidate for the solution of the PCP problem. All possible sequences of one-step expansions thus generate all possible solution candidates.

The theorem follows from the following claim, stating that a candidate solution is verified as a true solution only by finding a containment mapping from Q into the chase result of the corresponding expansion:

Claim. There is some view V generated by $\mathcal{P}$ such that

$$chase_{\mathcal{C}}(V) \sqsubseteq Q$$

iff V encodes a solution of the PCP problem. $\diamond$

Proof. Notice that the chase of any expansion E with the key constraint can only start at the common origin of the v- and w-chains, and can only continue as long as the labels in the chains situated at the same distance from the origin coincide. The chains are determined by a solution to the PCP problem if and only if they match on their entire length, which is equivalent to the chase proceeding to collapse the chains all the way to their ends. This is detected by the fact that the `l`- and the `r`-edges eventually share the same source, which in turn is the only way in which the query pattern can map into the chase result of E.
End of proof of claim.

■

This justifies our key-safety restriction, showing that it is maximally permissive. Theorems 4.6.2 and 4.3.2 immediately yield:

Corollary 4.6.1. *If $\mathcal{P}$ is recursive and not key-safe, then $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ is undecidable even if $\mathcal{C}$ consists of a single key constraint and $\mathcal{P}$ is a linear monadic program.*

Sets of full TGDs are trivially weakly acyclic, and yet we have:

Theorem 4.6.3. *If $\mathcal{P}$ is recursive, then* $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ contains only full TGDs and $\mathcal{P}$ is a monadic program.*

Proof: We use a reduction from PCP, adapting the construction from the proof of Theorem 4.6.2. The main difficulty here is to control that the two chains of v- and w-words are determined by the same sequence of integers, and that the chains match each other in length and labels. This was achieved in the proof of Theorem 4.6.2 by chasing with the key constraint.

We introduce fresh edge labels, $1, \ldots, n$, for n being the number of PCP words. We also use the labels *sync*, *end*, *up*, *down*.

We construct a monadic, (recursive) Datalog program $\mathcal{P}$, the set $\mathcal{C}$ comprising three families of TGDs, and a query Q such that the PCP problem has a solution iff $\textsc{Expr}^{\mathcal{C}}_{\mathcal{P}}(Q)$.

The Datalog program $\mathcal{P}$ contains:

- a rule for the distinguished IDB predicate *ans*: $ans() :- \ C(X)$;
- for every $1 \leq i \leq n$, assuming w.l.o.g. that $v_i = \alpha^i_1 \ldots \alpha^i_{k_i}$ and $w_i = \beta^i_1 \ldots \beta^i_{l_i}$, the rules

$$\begin{aligned} C(X) \ :- \quad & e(X, sync, X), \\ & e(X, \alpha^i_1, X_1), \ldots, e(X_{k_i-1}, \alpha^i_{k_i}, X_{k_i}), \\ & e(X, \beta^i_1, Y_1), \ldots, e(Y_{l_i-1}, \beta^i_{l_i}, Y_{l_i}), \\ & e(X, i, X_{k_i}), e(X, i, Y_{l_i}), \\ & C_v(X_{k_i}), C_w(Y_{l_i}); \end{aligned}$$

$$\begin{aligned} C_v(X) \ :- \quad & e(X, \alpha^i_1, X_1), \ldots, e(X_{k_i-1}, \alpha^i_{k_i}, X_{k_i}), \\ & e(X, i, X_{k_i}), C_v(X_{k_i}); \end{aligned}$$

$$\begin{aligned} C_w(X) \ :- \quad & e(X, \beta^i_1, Y_1), \ldots, e(Y'_{l_i-1}, \beta^i_{l_i}, Y_{l_i}), \\ & e(X, i, Y_{l_i}), C_w(Y_{l_i}); \end{aligned}$$

- the rules

$$\begin{aligned} C_v(X) &:- e(X, end, Y), e(Y, up, Z) \\ C_w(X) &:- e(X, end, Y), e(Y, down, Z) \\ & \quad e(X, a, X), e(X, b, X), \\ & \quad e(X, 1, X), \ldots, e(X, n, X), \\ & \quad e(X, sync, X) \end{aligned}$$

The program expands into chains that are not necessarily synchronized. We control synchronization by constraints. More precisely, we use TGDs to control that the two chains are determined by the same sequence of integers, and to control that the two chains match. $\mathcal{C}$ comprises:

- for each $1 \leq i \leq n$, the full TGD

$$\begin{aligned} & \forall X, Y, Z, T \\ & e(X, sync, Y) \wedge e(X, i, Z) \wedge e(Y, i, T) \rightarrow e(Z, sync, T) \end{aligned} \tag{4.21}$$

- for each $l, l' \in \{a, b\}$, the full TGD

$$\begin{aligned} & \forall X, Y, Z, T \\ & e(X, l, Y), e(X, l, Z), e(Y, l', T) \rightarrow e(Z, l', T) \end{aligned} \tag{4.22}$$

- for each $l \in \{a, b\}$, the full TGD

$$\begin{aligned} & \forall X, Y, Z, T, U, V, W \\ & e(X, l, Y), e(X, l, Z), e(Z, end, T), \\ & e(Y, end, V), e(T, up, U), e(V, down, W), \\ & e(Y, sync, Z) \rightarrow e(V, up, U) \end{aligned} \tag{4.23}$$

Intuitively, an expansion has an *end*-edge to signal the end of each chain, then an *up*-edge to signal the end of the chain of v-words, and a *down*-edge to signal the end of the chain of w-words.

The *sync* edges are added by the chase of the expansion with the family of TGDs (4.21), to mark the pairs of nodes on the two chains which represent chain prefixes determined by the same sequence of integers from $\{1, \ldots, n\}$.

Since the two chains of the expansion have a common origin (due to the expansion of IDB C), the chase with the family of TGDs (4.22) can only start at the origin, and continues down the chains only as far as the labels of the chain prefixes match. The two chains match entirely if and only if the chase with (4.22) stops at the chain ends (marked by *end*-edges).

If the chase with both families of TGDs (4.21) and (4.22) goes all the way to the end of the two chains, then both the sequence of integers and the sequence of labels coincide, hence the chains encode a PCP solution. This is detected by the family of TGDs (4.23), which apply only in that case, recording this fact by copying the *up*-edge from the end of the v-chain to the end of the w-chain, thus creating a node with both *up* and *down* edges emanating from it.

This is precisely what the query checks for:

$$\begin{aligned} Q : q() \quad :- \quad & e(T, a, T), e(T, b, T) \\ & e(T, 1, T), \ldots, e(T, n, T), \\ & e(T, \mathit{sync}, T), \\ & e(T, \mathit{end}, X), e(X, \mathit{up}, Y), e(X, \mathit{down}, Z) \end{aligned}$$

Similar to proof of Theorem 4.6.2, in order to enable mappings from the arbitrarily long chains of the expansions into the query, Q contains cycles into which every pair of chains can map. Indeed, it is easy to see that any expansion of $\mathcal{P}$ has a containment mapping into Q. Since the cycles in Q cannot map into the straight chains in the expansions of $\mathcal{P}$, the w-chain is ended by the cycles generated by the second rule of C_w.

It is easy to verify that Q can be mapped into the result of chasing some expansion of $\mathcal{P}$ with $\mathcal{C}$ iff the expansion encodes a PCP solution. ■

Corollary 4.6.2. *If $\mathcal{P}$ is recursive, then* $\textsc{Supp}^{\mathcal{C}}_{\mathcal{P}}(Q)$ *is undecidable even if $\mathcal{C}$ contains only full TGDs and $\mathcal{P}$ is monadic.*

Since INDs are a particular case of TGDs, it is interesting to contrast Theorem 4.6.3 and Corollary 4.4.3. Notice that there is no contradiction here, as weakly acyclic sets of INDs and sets of full TGDs have incomparable expressive power: weakly acyclic sets of INDs can express non-full TGDs, but INDs allow only one atom in the premise, while full TGDs allow multiple atoms.

4.7 Parameters

Our solutions to checking expressibility and support can be extended to the case when sources implement parameterized queries, expecting applications to provide the parameter values (recall Example 4.1.2).

There are two kinds of query evaluation plans one may adopt in the presence of parameters. The straightforward execution consists in the wrapper issuing in a first stage a series of service calls to the source without inspecting any intermediate results to determine how to instantiate parameters for the other calls. Once all call results come in, during the second stage the rewriting query is run over them and the result passed to the application query. This is the approach taken in [LRU96, LRU99]. We shall call this approach the *two-stage* evaluation. A more sophisticated evaluation strategy is based on the idea of interleaving query execution at the wrapper with service calls to the source. The evaluation of a subquery of the rewriting can thus provide parameter values for the subsequent calls needed by the non-evaluated part of the rewriting. This approach is used in [VP97] and, for finite sets of parameterized views, in [FLMS99], where it is known as the *dependent-join* evaluation.

If only two-stage evaluation is considered, there is an immediate reduction to the problem of non-parameterized views, based on the following observation:

Lemma 4.7.1. *In two-stage evaluation, for the views to be relevant to the problem of support or expressibility, their parameters must be filled in with constants appearing in the query or the source dependencies.*

This result follows immediately from Lemma 4.5.1 and generalizes a similar observation from [LRU96] to the presence of dependencies. It implies that it

suffices to generate a new program in which the parameters are replaced in all possible ways by the (bounded) set of constants in Q and $\mathcal{C}$, and test support and expressibility for the new program. In practice, an efficient implementation would extend the rewriting algorithm as suggested in [LRU96], by mapping parameters into constants from the query.

We next present expressibility and support under the more advanced dependent-join evaluation strategy. Our solution comes with no complexity overhead, in the sense that in the dependency-free case, our decision procedures have the same complexity as in the parameter-less case. This is non-trivial since the number of parameters that can occur in an expansion is a priori unbounded.

We start by introducing some auxiliary notions.

Notation. For parameters we adopt the $?X$ notation of [LRU96, LRU99], enabling the generation of parameterized views. We stress that by this notation, an input variable $?X$ will be considered different from some other variable X appearing in the same program rule.

Access patterns. An access pattern for a view $V(X_1, \ldots, X_k)$ is an expression α in $\{o, i\}^k$. We say that the X_j is an *output* (resp. *input*) variable if $\alpha(j) = o$ (resp. $\alpha(j) = i$). A view with access pattern α is denoted $V^\alpha(X_1, \ldots, X_k)$. Views generated by a Datalog program with parameters will be presented using this notation, by introducing an input head variable for each parameter.

Executable query. Notice that, for a (rewriting) query whose atoms have access patterns, there may not always be a way to satisfy the bindings for the input variables, i.e. the query is not executable. Following [NL04a, DLN05], we say that a query R formulated in terms of view names with binding patterns $\mathcal{V}$ is *executable* if the access patterns of R are such that every input variable appears first in an output position of some previous goal.

Expressibility / Support. We are now ready to extend the definitions of expressibility and support in the presence of parameterized views. We say that a query Q is *expressible* by a program $\mathcal{P}$ iff the query is equivalent to a query obtained from an expansion of $\mathcal{P}$ by replacing all input variables by constants. Note that this is the natural choice, since expressibility captures the cases in which a query

can be fully answered by just one "service call", without any post-processing. We say that Q is *supported* by $\mathcal{P}$ iff there exists an executable rewriting R using some finite set $\mathcal{V}$ of views with access patterns generated by $\mathcal{P}$.

Before going into the specific details, we first give a brief outline on how the solutions of the previous section can be extended to deal with parameterized programs. As before, we aim at reducing these problems to query answering using only a finite family of the specified views, defined by descriptors. First, since by the dependent-join mechanism input variables play an important role in how view goals interact in a rewriting, we need to keep track in descriptors of their query-view and view-query mappings. While this leads to descriptors of unbounded size (since the number of input variables is not bounded), we show that only a finite set of descriptors needs to be considered. Then, we extend algorithm **testSupport** to find an *executable* ordering of a rewriting in terms of descriptors. For this phase, we show that an expensive ordering search can be avoided, by relying on a canonical executable rewriting candidate. In conclusion, similar to the case without parameters, we obtain a sound, exponential-time algorithm for expressibility and support, which becomes complete in the absence of constraints or under restrictions on the interaction between program and constraints.

Modifying Example 4.1.1, the running example in this section is the following:

EXAMPLE 4.7.1. *Consider the schema from Example 4.1.1 extended with a relation airport(name) and the set of views specified by the parameterized Datalog program $\mathcal{P}''$, with 2 distinguished IDB predicates (ans_1 and ans_2):*

$$
\begin{aligned}
ans_1(A) &:- a(A) \\
ans_2(A,B) &:- f(A,C), ind(C,B) \\
ind(C,B) &:- f(C,C'), ind(C',B) \\
ind(C,?B) &:- f(C,?B), b(?B, \text{“}Paris\text{”}) \\
ind(C,?B) &:- f(C,?B), t(?B, \text{“}Paris\text{”})
\end{aligned}
$$

Note that the program differs from the one of Example 4.1.1 in two aspects: (a) the source admits direct access to the airport *relation (by ans_1) and (b) the destination of views concerning itineraries (by ans_2) is an input variable.*

Besides dependency (4.1)

$$\forall A, S \quad t(A,S) \wedge b(A,S) \longrightarrow s(A,S)$$

we assume the source verifies also the dependency

$$\forall A \quad b(A, \text{“}Paris\text{”}), t(A, \text{“}Paris\text{”}) \longrightarrow a(A) \tag{4.24}$$

which guarantees that any airport with a bus and train connection to Paris can be found in the airport *relation.*

Consider that the user asks the same query as in Example 4.1.1, i.e., itineraries of length 2 ending in an airport from which Paris is reachable by all the three transportation means

$$\begin{aligned}(Q) \quad q(A,B) \quad :- \quad & f(A,C), f(C,B), t(B, \text{“}Paris\text{”}), \\ & b(B, \text{“}Paris\text{”}), s(B, \text{“}Paris\text{”}).\end{aligned}$$

We recall that this query was supported in the setting of Example 4.5.1, as witnessed by the rewriting

$$(R) \qquad r(A,B) :- \quad V_1(A,B), V_2(A,B).$$

However, under the given access patterns, R no longer witnesses support since the conjunction of $V_1^{oi}(A,B)$ and $V_2^{oi}(A,B)$ is not executable. But by adding to this rewriting $U^o(B)$ as a first subgoal, where U is the one view generated by predicate ans_1, the query becomes executable:

$$(R') \qquad r(A,B) :- \quad U^o(B), V_1^{oi}(A,B), V_2^{oi}(A,B),$$

and moreover equivalent to Q. Indeed, the values for B are now passed by the dependent-join, and it can be easily checked that R' is equivalent to Q under the two dependencies, since the airport *goal of the rewriting maps in the result of chasing Q with* (4.24)*.*

We next discuss the decision procedure for view-based query answering using a *finite* set of parameterized views, under dependencies. This procedure will be then adapted to a finite set of view descriptors.

Answerable part. Given a query R formulated in terms of view names with access patterns $\mathcal{V}$, we call the *answerable part* of R (denoted $ans(R)$) the executable query with the same head as R and the body built one goal at a time from $body(R)$ as follows:

- start with an empty set of bounded variables, $\mathcal{B}$, then repeatedly
- find the first view goal $g^{\alpha}(\bar{X})$ in R not added to $ans(R)$ such that all the input variables of g are in $\mathcal{B}$; add this goal to $ans(R)$ and add its head variables $\bar{X}$ to $\mathcal{B}$.

Clearly, $ans(R)$ is an executable query and this procedure runs in quadratic time.

Executable Canonical Rewriting Candidate. Given a finite set of views with access patterns $\mathcal{V}$, an acyclic set of constraints $\mathcal{C}$, and a query Q, we call the *executable canonical rewriting candidate* of Q using $\mathcal{V}$ under $\mathcal{C}$, denoted $ECRC^{\mathcal{C}}_{\mathcal{V}}(Q)$, the query obtained as follows:

(i) compute $CRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ (as described in Section 4.5),

(ii) find its answerable part, $\mathbf{ans}(CRC^{\mathcal{C}}_{\mathcal{V}}(Q))$.

Similar to Lemma 4.5.1, results from [DLN05] guarantee the following:

Lemma 4.7.2 ([DLN05])**.** *If $\mathcal{V}$ is a set of parameterized views, then Q has a rewriting using $\mathcal{V}$ under $\mathcal{C}$ iff $ECRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ is itself one. This holds iff (a) $ECRC^{\mathcal{C}}_{\mathcal{V}}(Q)$ is safe and (b) $chase_{\mathcal{C}}(expand_{\mathcal{V}}(ECRC^{\mathcal{C}}_{\mathcal{V}}(Q))) \sqsubseteq Q$.*

EXAMPLE 4.7.2. *Revisiting Example 4.7.1, we know that the views V_1, V_2, U, generated (among others) by $\mathcal{P}''$, give an executable rewriting for Q, under $\mathcal{C} = \{(4.1), (4.24)\}$.*

Assume that an additional distinguished predicate is present in $\mathcal{P}''$, defined by the rule:

$$ans_3(A, B) \quad :- \quad f(A, ?C), ind(?C, B)$$

This rule generates, among others, two views that have the same subgoals as V_1 and V_2, but in which the intermediary stop is an input variable, too. Hence these views, denoted W_1 and W_2, will have one output and two inputs, their access pattern being (o, i, i).

Consider the set of views $\mathcal{V} = \{V_1, V_2, U, W_1, W_2\}$, which can all be mapped into $chase_{\mathcal{C}}(Q)$. By evaluating them on the body of $chase_{\mathcal{C}}(Q)$, we obtain the intermediary $CRC_{\mathcal{V}}^{\mathcal{C}}(Q)$:

$$\begin{aligned} R(A, B) \quad :- \quad & V_1^{oi}(A, B), V_2^{oi}(A, B), U^o(B), \\ & W_1^{oii}(A, C, B), W_2^{oii}(A, C, B) \end{aligned}$$

which is not executable since no value can be assigned to the C input variable. However, by computing $\mathbf{ans}(CRC_{\mathcal{V}}^{\mathcal{C}}(Q))$*, we eliminate the last two goals and reorder the rewriting, obtaining the $ECRC_{\mathcal{V}}^{\mathcal{C}}(Q)$:*

$$R(A, B) \quad :- \quad U^o(B), V_1^{oi}(A, B), V_2^{oi}(A, B)$$

which we know is indeed an executable rewriting of Q.

Testing expressibility and support. Similarly to the case without access patterns, we capture the usefulness of a view in the executable rewriting candidate by a descriptor, which takes also into account parameters and the access patterns they impose. Once the set of descriptors is obtained, checking expressibility amounts to checking if one of them denotes a view which becomes equivalent to Q after replacing input variables by constants. For testing support, we first construct the *partial rewriting candidate*, as described in Section 4.5. Since this candidate may not be executable, we need to compute its answerable part, which we call

the *executable partial rewriting candidate*. Finally, we check as in Corollary 4.5.1 whether this candidate is equivalent to Q under the dependencies, starting from the corresponding view fragments.

Finding descriptors. Since now we need to describe also the role of view goals in the answerable part of the rewriting candidate, we enrich the descriptor definition by taking into account input variables. More precisely, (a) input variables are treated as head variables, and (b) we add the corresponding access patterns to each descriptor, thus discriminating among views which are similar according to Definition 4.5.2 if they have distinct access patterns.

Definition 4.7.1. *For a query Q and a parameterized program $\mathcal{P}$, $E^{(p^{\alpha}(t),fr)}$ is called a* descriptor *w.r.t Q and $\mathcal{P}$ iff*

- *$E^{(p(t),fr)}$ is a descriptor in the sense of Definition 4.5.1, i.e. ignoring access patterns*
- *either p is not a distinguished predicate and α is empty, or p is a distinguished predicate and α is its corresponding access pattern.*

Definition 4.7.2. *Two descriptors (computed for a query Q and a parameterized program $\mathcal{P}$) $E_1^{(p_1^{\alpha}(t_1),fr_1)}$ and $E_2^{(p_2^{\beta}(t_2),fr_2)}$ are* similar *iff $E_1^{(p_1(t_1),fr_1)}$ and $E_2^{(p_2(t_2),fr_2)}$ satisfy the conditions of Definition 4.5.2 and their access patterns, α and β, are identical.*

EXAMPLE 4.7.3. *In the setting of Example 4.7.2, the descriptor for the view V_1^{oi} has, besides the components given in Example 4.5.2, the access pattern $\alpha_1 = (o, i)$. Similarly, the descriptor for the view W_1^{oii} has the components*

$$\begin{aligned} E_1 &= [f(Z_1, Z_2), f(Z_2, Z_3), t(Z_3, \text{“}Paris\text{”})] \\ p_1(t_1) &= ans(A, C, B) \\ fr_1 &= \{A : Z_1, C : Z_2, B : Z_3\} \\ \alpha_2 &= (o, i, i) \end{aligned}$$

One difficulty in extending descriptors in this way comes from the fact that there may be no bound on the number of input variables of generated views, leading to an unbounded number of descriptors and excluding any rewriting approach

based on descriptors. However, we know from [RSU95] that, *in the absence of constraints*, if a rewriting using views with binding patterns exists, then one with at most n (the number of variables of Q) distinct variables exists. This can in fact be extended to the case *with constraints*, showing that if a rewriting with a finite set of views exists, then there is also one in which the view atoms have at most n input variables, n being the number of variables of $chase_{\mathcal{C}}(Q)$. The intuition for this is that if a view with more than n parameters appears in a rewriting, then for sure some of those parameters will be bound to the same value. Hence it is sufficient to consider only descriptors with at most n inputs. Moreover, it was shown in previous work [DT03b], that if the constraints $\mathcal{C}$ are weakly acyclic, n is upper-bounded by a polynomial in the size of Q whose largest exponent depends only on $\mathcal{C}$.

Therefore, the procedure **findDescriptors** can easily be extended to take parameters into account. The bottom-up step will infer descriptors in which the binding pattern component may contain up to n distinct input variables and it will also record the access pattern α for a descriptor $E^{(p^{\alpha}(t),fr)}$. We obtain then a similar result to Lemma 4.5.2, using essentially the same proof.

Lemma 4.7.3. *If $\mathcal{C}$ is weakly acyclic, procedure **findDescriptors** extended with parameters outputs all pairwise dissimilar descriptors and is guaranteed to terminate in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$ and Q.*

To adapt procedure **testSupport** to parameterized programs, we only need to change the test for witnessing support to be based on Lemma 4.7.3 instead of Corollary 4.5.1. The new tests consist in building the executable partial rewriting candidate, testing that it is safe and that the query has a mapping into its expansion using the view fragments.

We can define decidability restrictions for parameterized views, as we did in Section 4.4 for the case without parameters. The definition of $\mathcal{C}$-locality is similar to Definition 4.4.1, but in addition to those conditions, we require that the views V and W have the same access patterns. For $\mathcal{C}$-independence of parameterized views, we reformulate Definition 4.4.2 to consider only queries R' and R that are executable.

The results of theorems 4.4.1 and 4.4.2, then also of theorems 4.5.1 and 4.5.2 extend to parameters.

Theorem 4.7.1. *If $\mathcal{C}$ is weakly acyclic, procedure* ***testSupport*** *for parameterized programs is a sound algorithm for checking support and runs in time exponential in the sizes of $\mathcal{P}$, $\mathcal{C}$ and Q. It becomes a complete decision procedure if $\mathcal{P}$ is a $\mathcal{C}$-local program generating $\mathcal{C}$-independent views.*

Proof: Compared to the case without parameters, the bound on the maximum number of descriptors is multiplied by a factor that is exponential in the maximum arity of the views (without increasing the complexity), but the rest of the soundness proof is the same as for Theorem 4.5.1.

The proof of completeness under restrictions is the same as in the proof of theorem 4.5.2, with the observation that, by the definition of $\mathcal{C}$-locality, the $\mathcal{W}$ have the same access patterns as the corresponding $\mathcal{V}$ views, hence the query over $\mathcal{W}$ is executable iff the one over $\mathcal{V}$ is. Thus we can still infer support of Q using $\mathcal{P}$ based on the support that uses $chase_{\mathcal{C}}(\mathcal{P})$. ■

4.8 Related Work

The necessity of describing infinite families of views exported by the source was first argued in [PGGMU95] and the problem of deciding support first solved (in the absence of constraints) in [LRU96, LRU99]. [LRU99] pioneers the idea of reducing support to rewriting the query using finitely many views. Views generated by the program are compared for *interchangeability*: V_1 and V_2 are interchangeable if in *every* rewriting R of Q, by replacing the V_1 goals with V_2 goals we still obtain a rewriting. [LRU99] shows that interchangeability induces finitely many equivalence classes on the set of all views generated by the program, and gives an algorithm to find one representative of each class. This finite set of representative views is then used to check for a rewriting. The resulting algorithm runs in doubly-exponential deterministic time. We can show however that interchangeability under dependencies yields infinitely many equivalence classes, thus precluding

the reduction from [LRU99] (see Example 4.A.1 in Appendix 4.A). Even in the absence of dependencies, we observe that interchangeability is unnecessarily strong, leading to a refinement of the view equivalence classes that yields exponentially more representatives than truly needed. Intuitively, instead of interchangeability in *every* rewriting of Q, the descriptor similarity condition (†) from Section 4.5 detects interchangeability with respect to only the *canonical* rewriting. This allows us to manipulate mapping/partial mapping pairs rather than *sets* thereof as in [LRU99], which yields the upper bound improvement from doubly-exponential to single-exponential time.

In the dependency-free setting, [VP00] improves the upper bound for support of [LRU99] to non-deterministic exponential time in the combined query and program size. However for practical purposes this still yields implementations that run in doubly-exponential time. In addition to the extension to constraints, our solution improves on [VP00] even in the dependency-free case, by achieving an exponentially better upper bound, proven to be essentially tight.

The problem of support strictly extends that of rewriting queries using finitely many views. The latter was treated in depth in the literature, considering various extensions pertaining to the language of queries and views [LMSS95b, ALM02, ACGP06, CNS06], to adding limited access patterns for the views [FLMS99, NL04b], to adding constraints (see the references in [DPT06]), and to mixing such extensions [DLN05]. The problem is NP-complete in the size of the query and views, in practice leading to deterministic exponential-time implementations, which is no better than for support. Prior work on information integration [LC00] studied answering queries using a finite set of views with limited access patterns with a different goal, namely finding maximally contained answers.

APPENDIX TO THE CHAPTER

4.A Interchangeability Is Unhelpful Under Dependencies

The following example shows that under dependencies, there are infinitely many equivalence classes of views with respect to interchangeability. This precludes the reduction described in [LRU99] from the problem of support to that of rewriting using finitely many views, as it involves focusing on representatives of the equivalence classes.

EXAMPLE 4.A.1. *We have a program $\mathcal{P}$ that produces unary views as follows:*

$$\begin{aligned}
V(X) &:- e(X,a,Y), C_r(Y)\\
C_r(X) &:- e(X,a,Y), C_r(Y)\\
C_r(X) &:- e(X,b,Y), e(Y',b,Y), e(Y',a,Y'),\\
&\quad\ e(Y,up,Z)\\
C_r(X) &:- e(X,b,Y), e(Y',b,Y), e(Y',a,Y'),\\
&\quad\ e(Y,down,Z)
\end{aligned}$$

Expansions are chains of a-labeled edges ending with a b-labeled edge and one of up *or* down*.*

Consider the query Q:

$$\begin{aligned}
Q() &:- e(D,a,D), e(D,b,A),\\
&\quad\ e(A,up,B), e(A,down,C)
\end{aligned}$$

The source obeys also one key constraint for each $l \in \{a,b\}$:

$$\forall X, Y', Y''\ e(X,l,Y'), e(X,l,Y'') \longrightarrow Y' = Y''.$$

We write V_n for the expansion with n a-labeled edges and ending with up*. We write U_n for the expansion with n a-labeled edges and ending with* down*.*

We can see that, for any n, the rewriting R_n defined as

$$R_n() :- V_n(X), U_n(X)$$

is an equivalent rewriting of Q.

However, replacing in R_n the V_n goal with any other view (V_i or U_i) would not yield another equivalent rewriting. So each V_n (and each U_n) is in its own equivalence class w.r.t. interchangeability in rewritings for Q. There are therefore infinitely many such equivalence classes.

Chapter 4 is currently being prepared for submission for publication of the material as it may appear in Theory of Computing Systems, 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 4, in part, is a reprint of the material as it appears in ICDT 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola.

Chapter 5

Compactly Encoded XML Views

ABSTRACT OF THE CHAPTER

This chapter addresses the problem of querying XML data sources that publish very large (potentially infinite) families of XPath queries. To compactly specify such families of data services I adopt the Query Set Specifications [PDP03] (QSS), a formalism close to context-free grammars.

I extend the definition of expressibility and support to QSS encoded services. A query Q is *expressible* by the specification $\mathcal{P}$ if it is equivalent to some expansion of $\mathcal{P}$. Q is *supported* by $\mathcal{P}$ if it has an equivalent rewriting using some finite set of $\mathcal{P}$'s expansions. I study the complexity of expressibility and support and identify large classes of XPath queries for which there are efficient (PTIME) algorithms. The study considers both the case in which the XML nodes in the results of the queries lose their original identity and the one in which the source exposes persistent node ids.

5.1 Introduction

We continue our investigation of expressibility and support and this time look at XML sources that publish a very large set of data services (possibly exponential in the size of the schema or even infinite), precluding explicit enumeration by the source owner as well as full comprehension by the client query developer.

The new technical challenge (compared to chapters 2 and 3) is that the system is responsible for automatically identifying and extracting from the compact encoding a finite set of data services–which we also call *views*–that can be used to answer the client query.

As mentioned in Chapter 4, this problem has been the object of several recent studies in a relational setting [LRU99, VP00, CDO09], but has not been addressed for sources that publish XML data (as is the case for most current Web Services). Since our focus is on practical algorithms, we consider sources that make XML data available through sets of views belonging to an XPath fragment for which the basic building blocks of rewriting algorithms, namely containment and equivalence, are tractable [MS04]. As a formalism for compactly representing large sets of such views, we adopt a variation of the Query Set Specification Language(QSSL) [PDP03], a grammar-like formalism for specifying XPath view families (see also [NÖ04] for its application to tree structured query interfaces).

In Chapter 3 we discussed the new opportunities for rewriting created by the recent industrial trend towards enhancing XPath queries with the ability to expose node identifiers and exploit them using intersection of node sets (via identity-based equality). In this chapter, we consider both the case in which the XML nodes in the results of the queries lose their original identity (hence a rewriting can only use one view) and the one in which the source exposes persistent node ids (and rewritings using multiple views are possible).

EXAMPLE 5.1.1. *Throughout this chapter we consider the example of a tourism agency that allows to find organized trips matching user criteria. The set of allowed queries are specified using a compact QSS encoding (to be described shortly, in Section 5.2). On the schema of the views published by the source, the client formulates a query q_1, asking for museums during a tour in whose schedule there is also a slot for taking a walk and which is part of a guided secondary trip:*

$$q_1\text{: } doc(T)//vacation//trip/trip[guide]//tour[schedule//walk]/museum$$

The system analyzes the query and the specification and finds two views that may be relevant for answering q_1. These are v_1, which returns museums in secondary trips for which there is a guide:

v_1: *doc(T)//vacation//trip/trip[guide]//museum*

and v_2*, which returns museums on a tour in which there has been also scheduled a walk:*

v_2: *doc(T)//vacation//trip//tour[schedule//walk]/museum*

q_1 *cannot be answered just by navigating into the result of* v_1 *or into the result of* v_2*. The reason is that* q_1 *needs both to enforce that the trip has a guide and that the tour has a walk in the schedule.* v_1 *or* v_2 *taken individually can enforce one of the two conditions, but not both, and navigation down into the view does not help either, since the output node museum is below the trip and tour nodes. Since no other views published by the source can contribute to rewriting* q_1*, in the absence of ids, the system will reject* q_1*, as it is neither expressed, nor supported by the source.*

However, if the views expose persistent node ids, we will show that q_1 *can be rewritten as an intersection of* v_1 *and* v_2*.*

XPath. We consider a tree pattern representation for XPath, as we defined it in Section 3.2

EXAMPLE 5.1.2. *Figure 5.1 shows the tree patterns corresponding to* v_1*,* v_2 *and* q_1 *from Example 5.1.1. Each node has a label and a unique node symbol, written inside parenthesis. Output nodes are distinguished in the graphical representation by a square.*

Contributions. We study the complexity of expressibility and support and identify large classes of XPath queries for which there are efficient (PTIME) algorithms. For expressibility, we give a PTIME decision procedure that works for any QSS and for any XPath query from a large fragment allowing child and descendant navigation and predicates. We show that support in the absence of ids remains in PTIME, for the same XPath fragment for which we studied expressibility. However, for this fragment, support in the presence of ids becomes coNP-hard. This is a consequence of previous results [CDO08], showing that rewriting XPath using an intersection of XPath views (a problem subsumed by support) is already coNP-hard. This is a major difference with respect to the relational case,

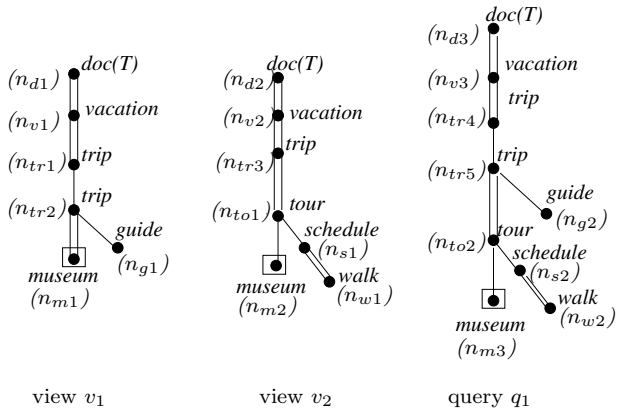


Figure 5.1: Tree patterns of queries v_1, v_2 and q_1

in which support and expressibility were proven inter-reducible [CDO09]. Since our focus is on practical algorithms, we propose a PTIME algorithm for id-based support that is sound for any XPath query, and becomes complete under fairly permissive restrictions on the query, without further restricting the language of the views. Our results are in stark contrast with previous results in the relational setting [LRU99, VP00], where already the simple language of conjunctive queries leads to EXPTIME completeness of expressivity and support [CDO09], but on the other hand is closed under intersection, which poses no additional problem.

Notation For the definitions of various XPath flavors we use, as well as tree patterns and operations on tree patterns, please consult Section 3.2. In addition, we introduce the following notation. We refer to main branch nodes of a pattern p by their *rank* in the main branch, i.e. a value in the range 1 to $|\mathrm{MB}(p)|$, for 1 corresponding to $\mathrm{ROOT}(p)$ and —$\mathrm{MB}(p)$— corresponding to $\mathrm{OUT}(p)$. For a rank k, by $p(k)$ we denote any pattern isomorphic with the subtree of p rooted at the main branch node of rank k. By $node_p(k)$ we denote the node of rank k in the main branch of p.

A *prefix* p' of a tree pattern p is any tree pattern that can be built from p by setting $\mathrm{ROOT}(p)$ as $\mathrm{ROOT}(p')$, setting some node $n \in \mathrm{MBN}(p)$ as $\mathrm{OUT}(p')$, and removing all the main branch nodes descendants of n along with their predicates. A *suffix* p' of a tree pattern p is any subtree of p rooted at a node in $\mathrm{MBN}(p)$.

We associate a name to each predicate in a pattern p (in lexicographic order). For a given predicate P, by n_P we denote the main branch node that is parent of P in q. By r_P we denote P's position on the main branch, i.e., the rank of the node n_P. By q_P we denote the pattern formed by the node n_P, as $\text{ROOT}(q_P)$, the pattern of P, and the edge connecting them. By $root_P$ we denote the node of p representing the root of P's pattern.

Chapter outline. The chapter is structured as follows. Section 5.2 describes the query set specifications (QSS). The problem of expressibility is analyzed in Section 5.3. The problem of support is studied starting from Section 5.4, first in the absence of persistent ids and then in their presence (Sections 5.5, 5.6, 5.7). QSS and rewriting language extensions are presented in Sections 5.8 and 5.9. Section 5.10 discusses related work.

5.2 Query Set Specifications

We consider sets of XPath queries encoded using a grammar-like formalism, Query Set Specifications (QSS), similar to [PDP03].

Definition 5.2.1. *A* Query Set Specification *(QSS) is a tuple* (F, Σ, P, S) *where*

- F *is the set of tree fragment names*
- Σ*, with* $\Sigma \cap F = \emptyset$ *is the set of element names*
- $S \in F$ *is the start tree fragment name*
- P *is a collection of expansion rules of the form*

  $f() \rightarrow tf \quad or \quad f(X) \rightarrow tf.$

  where f *is a tree fragment name,* tf *is a tree fragment and* X *denotes the output mark. Empty rules, of the form* $f \rightarrow$ *(no tree fragment) are also allowed.*

  f *is called the left-hand side (abbreviated as LHS) and*

  tf *is called the right-hand side (RHS) of the rule.*

A tree fragment is a labeled tree that may consist of the following:

- *element nodes, labeled with symbols from* Σ*,*

- *tree fragment nodes n labeled with symbols from F,*
- *edges either of child type, denoted by simple lines, or of descendant type, denoted by double lines,*
- *the output mark X associated to one node (of either kind).*

In any rule, in the RHS one unique node may have the output mark (X) if and only if that rule has the output mark on the LHS.

As a notation convention, we serialize QSS tree fragments as XP expressions with an output mark (X), if present.

QSS expansions. A finite expansion (in short *expansion*) of a QSS $\mathcal{P}$ is any tree pattern p having a body obtained as follows:

- starting from a rule $S(X) \rightarrow tf$,
- apply on tf the following expansion step a finite number of times until no more tree fragment names are left: for some node n labeled by a tree fragment name f, pick a rule defining f (i.e., f is the LHS) and replace n by the RHS of that rule; if n has the output mark, use only rules with LHS $f(X)$.
- set the node having the output mark as $\mathrm{OUT}(p)$.

We say that p is *generated* by $\mathcal{P}$. Note that the set of expansions can be infinite if the QSS is recursive.

Definition 5.2.2 (Expressibility and Support)**.** *For an XP query q, a QSS $\mathcal{P}$, and a rewriting language $\mathcal{L}_R$ we say that*

1. *q is* expressible *by $\mathcal{P}$ iff q is equivalent to an expansion of $\mathcal{P}$.*
2. *q is* supported *by $\mathcal{P}$ in $\mathcal{L}_R$ iff there is a finite set $\mathcal{V}$ of XP queries generated by $\mathcal{P}$, with corresponding view documents $D_{\mathcal{V}}$, such that there is a rewriting of q formulated in $\mathcal{L}_R$ that navigates only in documents from $D_{\mathcal{V}}$.*

The definition of support given above depends on the language $\mathcal{L}_R$ in which the rewritings can be expressed. If rewritings are expressed in XP , then all one can do is navigate inside one view. However, if the source exposes persistent node ids, it becomes possible to intersect view results. In this case, one can choose $\mathcal{L}_R$ to be $XP^{\cap}$ and use several views in more complex rewritings.

EXAMPLE 5.2.1. *The QSS $\mathcal{P}$ below generates queries returning information about museums that will be visited on a guided trip or as part of a tour in whose schedule there is also allotted time for taking a walk. Trips that appear nested are secondary trips.*

$$
\begin{aligned}
(\mathcal{P}) \qquad & f_0(X) \rightarrow doc(T)//vacation//f_1(X) \\
& f_1(X) \rightarrow trip/f_1(X) \\
& f_1(X) \rightarrow trip[guide]//museum(X) \\
& f_1(X) \rightarrow trip//tour[schedule//walk]/museum(X)
\end{aligned}
$$

It can be checked that v_1 and v_2 introduced before are among the expansions of $\mathcal{P}$. When considering v_1 and v_2 as user queries, we can also say they are expressed by $\mathcal{P}$.

Consider the following client query q_2, asking for museums that have temporary exhibitions and are visited in secondary trips:

$$q_2\text{: } doc(T)//vacation//trip/trip[guide]//museum[temp].$$

q_2 is obviously not expressed by $\mathcal{P}$ (there is no temp element node in $\mathcal{P}$). However, it is enough to filter the result of v_1 by the predicate [temp] to obtain the same result as q_2, hence q_1 is supported by $\mathcal{P}$:

$$q_2 \equiv doc(v_1)/v_1/museum[temp]$$

Consider the user query q_1 from Example 5.1.1. It can be checked that q_1 cannot be answered simply by navigating into a single view. Suppose now that the views expose persistent node ids. Using the algorithm from Chapter 4, the support of q_1 is witnessed by v_1 and v_2:

$$q_1 \equiv doc(v_1)/v_1/museum \cap doc(v_2)/v_2/museum.$$

Intuitively, this holds because q_1 is one of the interleavings of v_1 and v_2 and all other interleavings are contained in q_1.

Normalization For ease of presentation, we introduce additional normalization steps. First, the set of tree fragment names that have the output mark (denoted *unary*) is assumed disjoint from those that do not have it (denoted *boolean*). Second, we equivalently transform all rules such that, in any RHS, tree fragments

have depth at most 1, and the nodes of depth 1 can only be labeled by tree fragment names (i.e., a root and possibly some tree fragment children, connected by either /-edges or //-edges). For that, we may introduce additional tree fragment names. After normalization, for l being a label in Σ, $c_1, \ldots, c_n$, and $d_1, \ldots, d_m$ being two (possibly empty) lists of tree fragment names and g being a tree fragment name as well, any non-empty rule falls into one of the following cases:

$$\begin{aligned} f() &\rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()] \\ f(X) &\rightarrow l(X)[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()] \\ f(X) &\rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]/g(X) \\ f(X) &\rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]//g(X) \end{aligned}$$

For any fragment name f and rule

$$f(X) \rightarrow l[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()] \; edge \; g(X),$$

by v_f we denote any possible expansion of f via that rule. By v'_f we denote any pattern that can be obtained from the rule by (i) expanding g into the empty pattern, and (ii) expanding the c_is and the g_js in any possible way. Note that v'_f has only one main branch node.

EXAMPLE 5.2.2. *The result of normalizing the QSS $\mathcal{P}$ from Example 5.2.1 is the following specification:*

$$\begin{aligned} f_0(X) &\rightarrow doc(T)//f_1(X), & f_1(X) &\rightarrow vacation//f_2(X) \\ f_2(X) &\rightarrow trip/f_2(X), & f_2(X) &\rightarrow trip[guide]//f_5(X) \\ f_2(X) &\rightarrow trip//f_3(X), & f_3(X) &\rightarrow tour[f_4()]/f_5(X) \\ f_4() &\rightarrow schedule//f_6(), & f_5(X) &\rightarrow museum(X) \\ f_6() &\rightarrow walk \end{aligned}$$

5.3 Expressibility

We consider in this section the problem of expressibility: given a query q and a QSS $\mathcal{P}$ encoding a set of views, decide if there exists a view v generated by $\mathcal{P}$ that is equivalent to q.

Conceptually, in order to test expressibility, one has to enumerate the set of views and, for each view, check its equivalence to q. This is obviously unfeasible in our setting, since the set of views is potentially infinite. But the following observation delivers a naïve algorithm: only views that contain q have to be considered, and there are only finitely many distinct (w.r.t. isomorphism) candidates since containment mapping into q limits both the maximum length of a path (by the maximal path length in q) and the set of node labels (by the ones of q). Therefore, one can decide expressibility by enumerating all the candidate views and checking for each candidate if (a) it is equivalent to q, and (b) it is indeed an expansion of $\mathcal{P}$. However, this solution has limited practical interest beyond the fact that it shows decidability for our problem, since it is non-elementary in time complexity.

Our main contribution here is to provide a PTIME decision procedure for expressibility. The intuition behind our algorithm is the following. We do not enumerate expansions, and instead we group views and view fragments (which are assembled by the QSS to form a view) into *equivalence classes* w.r.t. their behavior in the algorithm for checking equivalence with q. Since there are fewer (only polynomially many) possible behaviors, manipulating such equivalence classes instead of explicit views or fragments thereof enables our PTIME solution.

As a compact representation for equivalence classes, we use *descriptors*. Informally, we use two kinds of descriptors for views or view fragments:

- *mapping descriptors*, which record if some expansion of a tree fragment name maps into a subtree of q,
- *equivalence descriptors*, which record if some expansion of a tree fragment name is equivalent to a subtree of q.

The rest of this section is organized as follows.

We first observe that equivalence for tree patterns is reducible to equivalence for a different flavor of patterns, *boolean tree patterns* ([MS04]). These are tree patterns of arity 0 (no output node) that test if evaluating a pattern over an XML document yields an empty result or not. Following this observation, for presentation simplicity, we solve expressibility for boolean tree patterns (Section 5.3.1).

Then, in Section 5.3.2, we show how expressibility for tree patterns (arity 1) can be reduced to expressibility for boolean tree patters.

5.3.1 Expressibility for boolean tree patterns

We study in this section expressibility for boolean tree patterns. Their semantics, based on the same notion of embedding, can be easily adapted from the case of arity 1: the result of applying a boolean tree pattern p to an XML tree t is either the empty set $\emptyset$ or the set $\{\text{ROOT}(t)\}$. In the first case, we say that the result is *false*, in the latter, we say it is *true*. Containment and equivalence for boolean tree patterns are also based on mappings, with the only difference that there is no output node.

In the remainder of this section all patterns (queries and views) are boolean tree patterns. A QSS will have either rules of the form

$$f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$$

or empty rules.

In order to clarify the role of descriptors and the equivalence classes they might stand for, let us first consider how one can test equivalence between a query q and view v. The classic approach for checking this is dynamic programming, bottom-up, using boolean matrices that bookkeep mappings in each direction: $M(n_1, n_2)$ is *true* if the subtree rooted at n_1 contains the one rooted at n_2.

We prefer instead a variation on this approach, which will enable our PTIME solution. Since wildcard is not used, equivalence between a query q and a view v translates into q and v being *isomorphic* modulo minimization. Assuming that q is already minimized, this means that v has to be q plus some redundant branches, i.e.

- q is isomorphic to (part of) v, i.e. there is a containment mapping ψ from q into v, and the inverse ψ^{-1} is a partial mapping from v into q,
- the partial mapping ψ^{-1} can be completed to a containment mapping from v into q

In the above, no two nodes of q can have the same image under ψ. In other words, some nodes of v have an "equivalence role", and there must be one such node corresponding to each node of q, while the remaining nodes are redundant and must only have a "mapping role". This suggests that it is enough to build bottom-up only one matrix M, for containment from subtrees of v into subtrees of q, if in in parallel we bookkeep in another matrix details about *equivalence* between subtrees. A field in the equivalence matrix, $E(n_1, n_2)$, for $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$, indicates that the subtree $v(n_1)$ is equivalent with the subtree $q(n_2)$.

With these two matrices, checking $v \equiv q$ by a bottom-up pass is straightforward, by applying the following steps until fix-point:

A) For each pair (n_1, n_2), $n_1 \in \text{NODES}(v)$, $n_2 \in \text{NODES}(q)$,
set $M(n_1, n_2)$ to *true* if:

1. for each /-child n of n_1 there exists a /-child n' of n_2 s.t. $M(n, n') = true$,
2. for each //-child n of n_1 there exists a descendant n' of n_2 s.t. $M(n, n') = true$.

B) Similarly, for each pair of nodes (n_1, n_2), $n_1 \in \text{NODES}(v)$,
$n_2 \in \text{NODES}(q)$, set $E(n_1, n_2)$ to *true* if:

1. for each /-child n of n_2 there exists a /-child n' of n_1 s.t. $E(n, n') = true$,
2. for each //-child n of n_2 there exists a descendant n' of n_1 s.t. $E(n, n') = true$,
3. for each /-child n of n_1 that was not referred to at step (1), there exists a /-child n' of n_2 s.t. $M(n, n') = true$,
4. for each //-child n of n_1 that was not referred to at step (2), there exists a descendant n' of n_2 s.t. $M(n, n') = true$.

We are now ready to present our PTIME algorithm for expressibility. We will adapt the above approach for testing equivalence, which builds incrementally (bottom-up, one level at a time) the mapping and equivalence details, to the setting when views are generated by a QSS by expanding fragment names. We will use *mapping* and *equivalence descriptors* to record for each tree fragment name if some

of its expansions witnesses equivalence with or existence of mapping into a part of the query. More precisely,

Definition 5.3.1. *For a fragment name f of a QSS $\mathcal{P}$, a* mapping descriptor *is a tuple $map(f, n)$, where $n \in \text{NODES}(q)$, indicating that f has an expansion v_f in $\mathcal{P}$ that root-maps in the subtree of q rooted at node n.*

An equivalence descriptor *is a tuple $equiv(f, n)$, where $n \in \text{NODES}(q)$, indicating that f has an expansion v_f in $\mathcal{P}$ that is equivalent with the subtree of q rooted at node n.*

For a fragment name f, a descriptor $equiv(f, n)$ will also tell us where the expansion it stands for maps (or not) in q. In other words, once we have an equivalence descriptor for a fragment name expansion, we can infer *all* mapping descriptors for it.

EXAMPLE 5.3.1. *Suppose that the data source publishes a modified version of the QSS from Example 5.2.2, enforcing the possibility of taking a walk on trips that contain tours. This translates into replacing the last rule for f_2 with the rule (unnormalized):*

$$f_2(X) \rightarrow trip[.//f_6()]//f_3(X).$$

A client interface generates and sends a query identical to v_2 of Example 5.1.2 to this source.

The proof of expressibility will consist in finding an equivalence descriptor for the root of the tree pattern. To infer the existence of this descriptor, we compute descriptors going bottom up in the pattern and in the normalized QSS from Example 5.2.2.

We start with the leaves, for which we find $d_1 = equiv(f_5, n_{m2})$ and $d_2 = equiv(f_6, n_{w1})$. Using d_2, we can infer the descriptor $d_3 = equiv(f_4, n_{s1})$, which, together with the descriptor for n_{w1}, enables a descriptor $d_4 = equiv(f_3, n_{to1})$. Since n_{w1} is a descendant of n_{tr3}, we can use d_2 and d_4 to build a descriptor $equiv(f_2, n_{tr3})$. This in turn enables a descriptor $equiv(f_1, n_{v2})$, which leads to inferring a descriptor for the root: $equiv(f_0, n_{d2})$.

Thus we can check that expressibility holds, even if v_2 is not isomorphic to any expansion of the QSS (since it has no predicate on the node labeled with trip).

Our algorithm for testing expressibility mimics the two steps (A) and (B) above, but applies them instead on QSS rules and fragment nodes via descriptors. Given descriptors for the fragment names in the RHS, we infer new descriptors for the fragment name on the LHS. The only notable difference with respect to the approach for checking equivalence is for steps (B.1) and (B.2). For a fragment name f and node $n \in \text{NODES}(q)$, fragment names children of f in a rule may have several *equiv* descriptors, referring to different nodes of q. We must choose one among them in a way that does not preclude the inference of a descriptor $equiv(f, n)$, when one exists. For that, we use a function tf-cover, which takes as input a set of nodes N, a set of tree fragment names C and an array L such that for every $n \in N$, $L(n) \subseteq C$. It returns *true* if there is a way to pick a distinct tree fragment name from each $L(n)$, for all $n \in N$. This function is based on a max-flow computation and its running time is $O((|C| + |N|) * |C|)$. We refer the reader to Appendix 5.A for the detailed definition of tf-cover.

The computation of descriptors (algorithm findDescExpr) starts with the productions without tree fragment nodes on the RHS. Then it continues to infer descriptors, using all productions, until a fixed point is reached. It runs in polynomial time because (a) there are only polynomially many descriptors (their number is proportional to the size of the QSS multiplied by the size of the query), and (b) each step for inferring a new descriptor runs in polynomial time.

Algorithm findDescExpr(q, $\mathcal{P}$)**:**

A. Start with an empty set of descriptors R.

B. For each rule $f() \rightarrow ()$, node $n \in \text{NODES}(q)$, add to R the
descriptor $map(f, n)$.

C. For each rule $f() \rightarrow l$ (i.e., the RHS has only one node) and
each node $n \in \text{NODES}(q)$ labeled by l, add to R the descriptors
$equiv(f, n)$ and $map(f, n)$.

Repeat until R unchanged:

D. For each rule $f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$,
add to R a descriptor $map(f, n)$ if n is labeled by l and

- for each fragment name c_i there exists a descriptor $map(c_i, n')$ s.t. n' is a /-child of n,
- for each fragment name d_j there exists a descriptor $map(d_j, n')$ s.t. n' is a descendant of n.

E. for each rule $f() \rightarrow l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()]$:
add to R the descriptors $equiv(f, n)$ and $map(f, n)$ if

1. tf-cover(N_1, C, L) returns *true*,
   where N_1 is the set of /-children of n, $C \subseteq \{c_1, \ldots, c_n\}$ is the set of fragment names that have a descriptor $equiv(c_i, n')$ for $n' \in N_1$ and, for each $n' \in N_1$, $L(n') \subseteq C$ is the set of fragments names that have a descriptor $equiv(c_i, n')$.
2. tf-cover(N_2, D, L) returns *true*,
   where N_2 is the set of //-children of n, $D \subseteq \{d_1, \ldots, d_m\}$ is the set of fragment names that have a descriptor $equiv(d_j, n')$ for $n' \in N_2$ and, for each $n' \in N_2$, $L(n') \subseteq D$ is the set of fragments names that have a descriptor $equiv(d_j, n')$.
3. for each fragment name $c_i \notin C$, there exists a descriptor $map(c_i, n')$ s.t. n' is a /-child of n,
4. for each fragment name $d_j \notin D$ there exists a descriptor $map(d_j, n')$ s.t. n' is a descendant of n.

We can prove the following:

Theorem 5.3.1. *A boolean tree pattern q is expressed by a QSS $\mathcal{P}$ iff* findDescExpr*$(q, \mathcal{P})$ outputs a descriptor $equiv(S, \mathrm{ROOT}(q))$, for S being the start fragment name of $\mathcal{P}$.* findDescExpr *runs in polynomial time in the size of the query and of the QSS.*

Proof: [Sketch] If the input query q is minimized, an expansion equivalent to it is made of a pattern isomorphic to q plus some additional branches or sub-branches that all map into q. Equivalence descriptors witness the parts that are

isomorphic to q. Mapping descriptors guarantee that the additional predicates (or sub-predicates) also map into q. We can prove by induction on the length of the longest path in q that the algorithm outputs a descriptor an equivalence descriptor (for the root of q) $equiv(S, \text{ROOT}(q))$ if and only if $\mathcal{P}$ expresses q. ■

Remark. The assumption that the input query q is minimized is important for our algorithm. It allows us to avoid a bottom-up approach that might also have to bookkeep mappings from the query into views. That would require descriptors that pair *a set of subtrees* from q with an expansion, yielding a descriptor space of exponential size. Since no two nodes of q can have the same image under the ψ function described above, it suffices to use simpler descriptors, that relate only one subtree of q with an expansion.

5.3.2 Expressibility for tree patterns

We now consider expressibility for standard tree pattern queries (patterns with an output node).

It is well known from previous literature that one can reduce problems such as tree pattern containment and equivalence to containment, respectively equivalence, for boolean patterns. This is based on the following translation: let s be a new label (denoting *selection*), and for a tree pattern p let p_0 denote the boolean tree pattern obtained from p by (i) adding a /-child labeled s below the output node of p, and (ii) removing the output mark. From [MS04], for two tree patterns p and p', we have that $p \equiv p'$ iff $p_0 \equiv p'_0$.

A similar transformation can be applied for expressibility. Given a QSS $\mathcal{P}$, let $\mathcal{P}_0$ be the QSS obtained from $\mathcal{P}$ by (i) plugging a /-child labeled s below each node having an explicit label and the output mark, and (ii) making all rules and tree fragment names boolean by removing their output mark. (Since $\mathcal{P}$'s sets of unary and boolean tree fragment names were assumed disjoint, the newly obtained QSS $\mathcal{P}_0$ generates only boolean tree patterns having exactly one s-labeled node.) Using Theorem 5.3.1, we can immediately prove the following:

Theorem 5.3.2. *A tree pattern query q is expressed by a QSS $\mathcal{P}$ iff the boolean tree pattern q_0 is expressed by the QSS $\mathcal{P}_0$.*

5.4 Support

For the problem of support, the fact whether the source enables persistent node ids (that are then exposed in query results) or not has a significant impact on the rewrite plans one can build. In both settings, with or without node ids, rewriting under an explicitly listed set of views has been studied in previous literature. We will now revisit them for support.

In the first setting, the identity of the nodes forming the result of a query is not exposed in query results. By consequence, the only possible rewrite plans consist in accessing a view result and maybe navigating inside it (via query *compensation*). This setting was considered in [XÖ05], and the rewriting problem was shown to be in PTIME for XP . We study support in the absence of ids in Section 5.4.1. Our main result is that support reduces to expressibility, which allows us to reuse the PTIME algorithm given in Section 5.3.

In the second setting, for which rewriting under an explicit set of views was studied in Chapter 4, data sources expose persistent node ids. This enables more complex rewrite plans, in which the *intersection* of view results plays a crucial role. We revisit this setting, for the support problem, in Section 5.5. As our general approach, we will apply the same kind of reasoning that was used for expressibility. We will group views into equivalence classes w.r.t. crucial tests for support and we will manipulate classes (encoded as *view descriptors*) instead of explicit views. This will enable us to avoid the enumeration of a potentially large space of views and rewrite plans.

5.4.1 Support in the absence of ids

When persistent identifiers are not exposed, a rewrite plan consist in accessing a view's result and maybe navigating inside it, and this navigation is called *compensation*. This is why expressibility and support in the absence of ids remain strongly related, as support simply amounts to finding a candidate view v which, via compensation, becomes equivalent with the input query.

Let us first introduce as notation for this operation the compensate func-

tion, which performs the concatenation operation from [XÖ05], by copying extra navigation from the query into the rewrite plan. For a view $v \in XP$, an input query q, and a main branch rank k in q, compensate(v, q, k) returns the query obtained by deleting the first symbol from $x = xpath(q(k))$ and concatenating the rest to v. For instance, the result of compensating $v =$ a/b with $x =$ b[c][d]/e is the concatenation of a/b and [c][d]/e, i.e. a/b[c][d]/e.

We can reformulate the result from previous literature as follows:

Theorem 5.4.1 ([XÖ05])**.** *Given a set of explicit views $\mathcal{V}$, a query q can be answered by $\mathcal{V}$ if and only if there exists a view v and main branch rank k in q such that* compensate$(v, q, k) \equiv q$.

Going now to views encoded as QSS expansions, we will reduce the problem of support to expressibility, following the idea that support amounts to expressibility by a certain "compensated" specification.

From a given QSS $\mathcal{P}$, we will build a new QSS that generates, besides $\mathcal{P}$'s expansions, all their possible compensated versions w.r.t. q. More precisely, given an input query q and a QSS $\mathcal{P}$, let $comp(\mathcal{P}, q)$ denote the QSS obtained from $\mathcal{P}$ as follows:

For any rule yielding the output node, i.e., of the form

$$f(X) \rightarrow l(X)[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()],$$

for each rank k in q, add a new rule, of the form (with a little departure from the normalized QSS syntax):

$$f(X) \rightarrow \text{compensate}(l[c_1(), \ldots, c_n(), .//d_1(), \ldots, .//d_m()], q, k)$$

We can prove the following:

Theorem 5.4.2. *A query q is supported by a QSS $\mathcal{P}$ if and only if it is expressed by the QSS $comp(\mathcal{P}, q)$.*

EXAMPLE 5.4.1. *An example of support in the absence of persistent ids has already been given in Example 5.2.1: q_2 can be rewritten by compensating v_1 with a temp predicate.*

5.5 Support in the presence of Ids

We consider in this section the problems of support in the presence of node ids, denoted in the following *id-based support*. First, deciding the existence of a rewriting for an XP query under an explicit set of XP views becomes coNP-hard, as it was shown in Chapter 4.

Since our focus is on efficient algorithms for support, we next investigate the tightest restrictions for tractability. We consider the fragment of *extended skeletons* (XP_{es}), for which the rewriting problem was shown to be tractable in [CDO08]. The restrictions imposed by the XP_{es} fragment on the input query were shown to be necessary for tractability, as their relaxation leads to coNP-hardness. It is therefore natural to ask whether the support problem is also tractable for input queries from this fragment. Note that one cannot do better, i.e., obtain a decision procedure for input queries outside this fragment, since the problem of support subsumes the rewriting problem.

The remainder of this chapter is dedicated to studying support for extended skeletons, focusing on efficient (PTIME) solutions that are sound in general (i.e., for any XP input query) and complete under fairly general conditions, and this *without restricting the language of views* (which remains XP). We show that id-based support exhibits a complexity dichotomy: the sub-fragment of XP_{es} representing queries that have at least one //-edge in the main branch, denoted hereafter *multi-token*, continues to be in PTIME (Theorem 5.6.5), but the complementary sub-fragment that represents queries with only /-edges in the main branch, denoted hereafter *single-token*, interestingly, is NP-hard (see Theorem 5.7.1).

The fragment of multi-token queries is particularly useful in practice since often, for reasons such as conciseness or generality in the presence of schema heterogeneity, one does not want to write queries that specify all the navigation steps in a document.

Extended skeletons (XP_{es}). We will focus on the language XP_{es}, defined in Section 3.3.2. In short, this fragment limits the use of // in predicates, as follows: a main branch node n of a pattern p will not have predicates that may become redundant in some interleaving p might be involved in. For instance, ex-

pressions such as a[b//c]/d or a[b//c//d]/e/d are in XP_{es}, while a[b//c]/b, a[b//c]//d, or a[//b]/c are not. XP_{es} does not restrict in any way the usage of //-edges in the main branch or the usage of predicates with /-edges only.

We study in Section 5.6 support for multi-token queries and, in Section 5.7, support for single-token queries.

5.6 Multi-token queries

We consider in this section id-based support for XP_{es} multi-token queries. For presentation simplicity, we first limit the discussion to rewrite plans that are intersections of views (no compensation before the intersection step). The extension to general $XP^{\cap}$ plans, i.e., intersections of (possibly compensated) views, is detailed in Section 5.6.4.

As in the case of expressibility, we think of views as grouped into equivalence classes w.r.t. to crucial tests for support. We manipulate such classes, which are represented by *view descriptors*, instead of explicit views, avoiding the enumeration of a potentially large space of views and rewrite plans. Given that a QSS constructs views by putting together fragments, we have to construct our view descriptors from *fragment descriptors*, which represent equivalence classes for fragment expansions.

This section is organized as follows. In order to clarify the role of view descriptors and the equivalence classes they stand for, we first revisit in Section 5.6.1 the PTIME algorithm from Chapter 4 for deciding if an XP_{es} multi-token query q can be rewritten by an intersection of explicit XP views $\mathcal{V}$ already known to contain q. That algorithm was based on applying DAG-pattern rewrite steps towards a tree pattern and then checking equivalence with q. We reformulate it into an algorithm (testEquiv) that applies individual tests on the view definitions instead. Then, in Section 5.6.2, we introduce equivalence classes for views w.r.t. the tests of testEquiv, and *view descriptors* as a means to represent such classes. We reformulate the testEquiv algorithm into a new algorithm, testEquivDesc, that runs on view descriptors instead of explicit view definitions. Finally, in Section 5.6.3

we give a PTIME sound and complete algorithm for computing descriptors for the expansions of a QSS.

5.6.1 Rewriting with an explicit set of views

Let the input multi-token query q be of the form $q = ft//m//lt$ where ft denotes the first token, lt denotes the last token and m denotes the intermediary part (m may be empty). Let $\mathcal{V} = \{v_1, \ldots, v_k\}$ denote a set of views such that $q \sqsubseteq v_i$ for each view v_i. Let each view v_i be of the form $v_i = ft_i//m_i//lt_i$.

Notation. Let $ft_{\mathcal{V}}$ denote the query obtained by "combining" $ft_1, \ldots, ft_n$ as follows: start by coalescing their roots, then continue by coalescing top-down any pair of main branch nodes that have the same parent and the same label. This process yields a tree pattern because each first token ft_i maps in the first token of q, ft, hence each $\mathrm{MB}(ft_i)$ is a prefix of $\mathrm{MB}(ft)$. Let $lt_{\mathcal{V}}$ denote the query obtained by "combining" $lt_1, \ldots, lt_n$ similarly: start by coalescing the output nodes, then continue by coalescing bottom-up any pair of main branch nodes that have a common child and the same label.

EXAMPLE 5.6.1. *For instance, for two views $\mathcal{V} = \{v', v''\}$,*

$$\begin{aligned} v' &= doc(T)/vacation/trip[guide]//tour/museum, \\ v'' &= doc(T)/vacation[//walk]//museum[gallery], \end{aligned}$$

the result of combining their first tokens, respectively last tokens is

$$\begin{aligned} ft_{\mathcal{V}} &= doc(T)/vacation[//walk]/trip[guide], \\ lt_{\mathcal{V}} &= tour/museum[gallery]. \end{aligned}$$

Given $\mathrm{MB}(ft)$, $\mathrm{MB}(lt)$, if there exists a minimal (non-empty) prefix pr_q of $\mathrm{MB}(lt)$ that is isomorphic with a suffix of $\mathrm{MB}(ft)$, let $\mathrm{MB}(lt)'$ denote the pattern obtained from $\mathrm{MB}(lt)$ by cutting out pr_q. Then, let l_q denote the linear pattern $\mathrm{MB}(ft)/\mathrm{MB}(lt)'$. If l_q is undefined by the above (i.e., there is no pr_q), by convention it is the empty pattern.

EXAMPLE 5.6.2. *For instance, for the query*

$$q = doc(T)/vacation[//walk]/tour//tour/museum,$$

l_q is well-defined, as $l_q = doc(T)/vacation/tour/museum$.

Given MB(ft) and MB(m), if there exists a minimal (non-empty) suffix of MB(ft) that is isomorphic with a prefix of MB(m), let MB(ft)$_m$ denote the pattern obtained from MB(ft) by cutting out this suffix. If MB(ft)$_m$ is undefined by the above, by convention it is the empty pattern. Similarly, given MB(lt) and MB(m), if there exists a minimal (non-empty) prefix of MB(lt) that is isomorphic with a suffix of MB(m), let MB(lt)$_m$ denote the pattern obtained from MB(lt) by cutting out this prefix. If MB(lt)$_m$ is undefined by the above, by convention it is the empty pattern.

We are now ready to present our reformulation of the PTIME algorithm of Chapter 3, which will test that $\cap\mathcal{V} \sqsubseteq q$. By Lemma 3.2.4, q must contain each possible interleaving i of the set $\mathcal{V}$ or, in other words, for each $i \in interleave(\mathcal{V})$ the following should hold:

- the first token of q can be mapped in the first token of i s.t. the image of ROOT(q) is ROOT(i),
- the last token of q can be mapped in the last token of i s.t. the image of OUT(q) is OUT(i),
- the images of these two tokens in i are disjoint,
- the intermediary part m (if non-empty) of q can be mapped somewhere between the two images in i.

We can prove the following:

Theorem 5.6.1. *For an XP query q and a set of XP views $\mathcal{V}$,* testEquiv *is a sound PTIME procedure for testing $q \equiv \cap\mathcal{V}$.*

Proof: [Sketch] The first condition ensures that any interleaving i starts by a /-pattern into which ft has a containment mapping and ends by a /-pattern into which lt has a containment mapping.

Let us now consider the case when q's intermediary part m is empty, i.e., q is of the form $ft//lt$.

Algorithm 1 testEquiv($\mathcal{V}, q$)

1: **let each** $v_i = ft_i//m_i//lt_i$
2: **let** $q = ft//m//lt$
3: **begin**
4: **compute** the patterns $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$
5: **compute** the pattern l_q
6: **compute** the patterns $\mathrm{MB}(ft)_m$ and $\mathrm{MB}(lt)_m$
7: **if** $ft_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv lt$ **then**
8: **if** m is empty **then for each** $v_i \in \mathcal{V}$
9: **if** $\mathrm{MB}(v_i)$ does not map into l_q **then** output **true**
10: **else** (m non-empty) **for each** $v_j \in \mathcal{V}$
11: **if** v_j can be seen as $prefix_j//m'//suffix_j$ **s.t.**
12: $m' \equiv m$
13: $\mathrm{MB}(prefix_j)$ does not root-map into $\mathrm{MB}(ft)_m$
14: $\mathrm{MB}(suffix_j)$ does not output-map into $\mathrm{MB}(lt)_m$
15: **then** output **true**
16: **end**

In this case, condition (line 9) guarantees that in any interleaving i the images of ft and lt (by the containment mappings mentioned above) are disjoint (i.e., they do not map into the same token): If l_q is the empty pattern, this is immediate. Otherwise, since $l_q \not\sqsubseteq \mathrm{MB}(v_i)$, this means that (a) no interleavings with main branch l_q can be built, and furthermore (b) no interleavings with an even shorter main branch (that would be obtained by cutting a bigger prefix from $\mathrm{MB}(lt)$) can be built either. By the fact that these two containment mappings have disjoint images, their union yields a containment mapping from q into i, hence $i \sqsubseteq q$.

We now consider the case when m is not empty.

For this case, besides the fact that in any interleaving i the images of ft and lt must be disjoint, the rest of q (the m part) must also map somewhere between these images. All this is guaranteed by the conditions of lines 11-14.

First, v_j has a sub-query m' which, considered in isolation, is equivalent (i.e.

isomorphic modulo minimization) with m. Then, conditions (lines 13-14) imply that in any interleaving i of the views, nodes from the m' part of v_j cannot be collapsed with nodes from the first or last tokens of the various views. More precisely, they imply that the minimal prefix (resp. suffix) of MB(lt) (resp. MB(ft)) cannot be collapsed with the part of MB(m') to which it is isomorphic (by the definition of MB$(ft)_m$ and MB$(lt)_m$). By the minimality property, if there are some coalescing opportunities, the ones that are ruled out here must be among them. Hence the part $ft_{\mathcal{V}}$ (by which i starts) and the part $lt_{\mathcal{V}}$ (by which i ends) are disjoint, and there are at least $|m'|$ main branch nodes in between.

Then, the rest of q, m, will also map in between, since m maps in any pattern resulting from the interleaving of m' with other view parts (we can compose the mapping from m to m' with the onto function by which i is built). It follows easily that q has a containment mapping into any interleaving i of $\cap\mathcal{V}$.

We now consider how one can verify conditions (lines 9, 11-14) in polynomial time. For (line 9), the non-existence of a containment mapping between two linear paths could be easily translated into a containment mapping test.

Then, conditions (lines 11-14) amount to the following:

- find the views that have a sub-query equivalent to m (an equivalence test) and, for each of them,
- check the non-existence of the two mappings (even though $prefix_j$ root-maps into ft, hence MB($prefix_j$) also root-maps into MB(ft), and $suffix_j$ output-maps into lt, hence MB($suffix_j$) also output-maps into MB(ft)).

The first item is immediate. Then, for lines 13-14, since we are dealing again with linear patterns, testing if the two mappings fail can be done using a bottom-up (in the case of MB($suffix_j$)) respectively top-down (in the case of MB($prefix_j$)) procedure as the one described in Section 5.6.3, advancing one token at a time. ■

For XP_{es} multi-token queries, we can also prove completeness:

Theorem 5.6.2. *For an XP_{es} multi-token query q and a set of XP views $\mathcal{V}$,* testEquiv *is complete for testing $q \equiv \cap\mathcal{V}$.*

Proof: The proof is essentially a reformulation of the part of the proof of Theorem 3.3.5 that concerns multi-token queries. The difference is that testEquiv does not use rewriting rules, but verifies directly conditions that are necessary for the existence of a rewriting.

Let us go through all the checks performed by testEquiv. The test at Step 7 is necessary because, as we saw in Chapter 3, if there is a rewriting, it can be obtained by collapsing the view patterns. And the final result of collapsing them would need to have the same main branch as the query.

If q has only 2 tokens (i.e., the middle part m is empty), then $\mathrm{MB}(q)$ is of the form $\mathrm{MB}(ft)//\mathrm{MB}(lt)$. We saw in the proof of Theorem 5.6.1 that the test at line 9 is sufficient for the existence of a rewriting. If the test does not hold, then it means that l_q cannot be empty, otherwise no view could map into it. It also means that all view main branches are shorter in length than $\mathrm{MB}(q)$, hence there is an interleaving i with $|\mathrm{MB}(i)| < |\mathrm{MB}(q)|$. This implies that q does not map into i, hence q cannot have a rewriting using the intersection of the views.

Suppose that q has more than 2 tokens (m is non-empty). Let $\{v_i'\}$ be the patterns obtained from $\mathcal{V}$ by replacing, for each view v_i its first token with $ft_{\mathcal{V}}$ and its last token with $lt_{\mathcal{V}}$. By Lemma 3.A.4, the DAG d formed by intersecting all v_i' is union-free iff there is a v_j' (obtained from a view v_j), with a middle part mid', such that the middle parts of all other views map into it. If d is not union-free, then there is no rewriting, because d cannot be equivalent to q. Hence d is union-free, and its middle part is mid'. If $d \equiv q$, if follows that $m \equiv m'$, where m' is a subpattern of mid'. Since mid' is a subpattern of v_j, m' is also a subpattern of v_j.

Suppose that $\mathrm{MB}(\mathit{prefix}_j)$ root-maps into $\mathrm{MB}(ft)_m$. This also means that $\mathrm{MB}(ft)_m$ is not empty, hence there is an overlap between $\mathrm{MB}(ft)$ and $\mathrm{MB}(m)$. By the way $ft_{\mathcal{V}}$ was constructed, there must be a view v_k whose first token has the main branch as $ft_{\mathcal{V}}$. Since $ft_{\mathcal{V}} \equiv ft$, there is at least one interleaving i starting with $\mathrm{MB}(ft)_m$ and continuing with m' (we say that it is an interleaving that collapsed the overlapping parts betwen $\mathrm{MB}(ft)$ and $\mathrm{MB}(m')$). But q does not map into i because there is no mapping for the suffix of ft that follows after $\mathrm{MB}(ft)_m$.

Similarly, we can show that there is a rewriting only if $\mathrm{MB}(\mathit{suffix}_j)$ does not

output-map into $\mathrm{MB}(lt)_m$. ■

5.6.2 View descriptors

We detail in this section how one can perform the tests of algorithm testEquiv even when abstracting away from the view definitions. The key idea is that one does not need the complete definitions but only some particular details touched by these tests. With respect to these details, views can be seen as grouped into equivalence classes and views from the same class will be equally useful in the execution of the algorithm. This idea will be exploited by our *view descriptors*. We then reformulate testEquiv in terms of view descriptors in algorithm testEquivDesc. More precisely, assuming we are dealing with expansions of a QSS $\mathcal{P}$ with start fragment name S,

For line 7 **of** testEquiv**.** For the part $ft_{\mathcal{V}} \equiv ft$: a *first-token descriptor* will be a tuple $\mathbf{ft}(\mathbf{S}, \mathbf{p})$, where p denotes any pattern that can be built from q's first token ft by removing all its predicates, except eventually for one. Such a descriptor indicates that there exists an expansion v s.t. $q \sqsubseteq v$ and v's first token is of the form p, plus eventually other predicates (ignored in the descriptor). These descriptors represent partitions (equivalence classes) of the space of views containing q w.r.t. their first tokens and the predicates on them. Each view will belong to at least one such class, but may belong to several (for different choices of predicates).

Similarly, for the part $lt_{\mathcal{V}} \equiv lt$: a *last-token descriptor* is a tuple $\mathbf{lt}(\mathbf{S}, \mathbf{p})$, where p denotes any pattern that can be built from q's last token lt by removing all its predicates, except eventually for one. Such a descriptor indicates that there exists an expansion v s.t. $q \sqsubseteq v$ and v's last token is of the form p, plus eventually other predicates (ignored in the descriptor).

It is easy to see that the **ft** and **lt** view descriptors allow us to compute the patterns $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$, provided they verify $ft_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv lt$, without requiring the actual first and last tokens. The domain of these descriptors is quadratic in the size of q.

For line 9 **of** testEquiv**.** An *l-descriptor* is a tuple $\mathbf{l}(\mathbf{S})$, indicating that there exists an expansion v verifying $q \sqsubseteq v$ and $l_q \not\sqsubseteq \mathrm{MB}(v)$. (This type of descriptor is

an alias for the condition of line 9, denoting a partition of the space of views into two complementary equivalence classes.)

For lines $11-14$ **of** testEquiv**.** An *m-descriptor* is a tuple $\mathbf{m}(\mathbf{S})$, indicating that there exists an expansion v verifying $q \sqsubseteq v$ and all the conditions of lines $11-14$. (This type of descriptor is also an alias, denoting a partition of the space of views into two complementary equivalence classes.)

We are now ready to reformulate testEquiv into an algorithm that runs directly on the set of view descriptors $\mathcal{D}$, instead of the explicit views $\mathcal{V}$ to which they correspond. Unsurprisingly, the new algorithm follows closely the steps of testEquiv, since the descriptors are tailored to its various tests.

Algorithm 2 testEquivDesc($\mathcal{D}$, q)

1: **begin**
2: **for all** descriptors $\mathbf{ft}(\mathbf{S}, \mathbf{p}) \in \mathcal{D}$
3: **compute** the pattern $ft_{\mathcal{V}}$
4: **for all** descriptors $\mathbf{lt}(\mathbf{S}, \mathbf{p}) \in \mathcal{D}$
5: **compute** the pattern $lt_{\mathcal{V}}$
6: **if** $lt_{\mathcal{V}} \equiv ft$ and $lt_{\mathcal{V}} \equiv ft$ **then**
7: **if** m is empty **then**
8: **if** there exists a descriptor $\mathbf{l}(\mathbf{S}) \in \mathcal{D}$ **then** output **true**
9: **else if** there exists a descriptor $\mathbf{m}(\mathbf{S}) \in \mathcal{D}$ **then** output **true**
10: **end**

We can prove the following:

Theorem 5.6.3. *For an XP query q, a finite set of XP views $\mathcal{V}$ and their corresponding descriptors $\mathcal{D}$,* testEquiv*$(q,\mathcal{V})$ outputs* true *if and only if* testEquivDesc*$(q, \mathcal{D})$ does so.*

Proof: The fragments of patterns in the set of **ft** and **lt** descriptors computed by testEquivDesc contains all the main branches and predicates of first and last tokens of views. It follows immediately that testEquiv and testEquivDesc compute

the same $ft_{\mathcal{V}}$ and $lt_{\mathcal{V}}$ patterns. The tests at lines 8–9 are trivially equivalent, by the definition of **l** and **m** descriptors. ■

EXAMPLE 5.6.3. *For the query q_1 in Example 5.1.1, $ft = doc(T)$, $m = vacation//trip/trip[guide]$ and $lt = tour[schedule//walk]/museum$.*

For the QSS $\mathcal{P}$ from Example 5.2.1 and its two expansions v_1 and v_2, v_1 can be represented by the descriptors $\mathbf{ft}(S, doc(T))$, $\mathbf{lt}(S, museum)$ and, since v_1 has the required form $pref_1//m//suff_1$, by $\mathbf{m}(S)$ too.

Similarly, v_2 can be represented by the descriptors $\mathbf{ft}(S, doc(T))$ and $\mathbf{lt}(S, tour[schedule//walk]/museum)$.

Running on these descriptors, testEquivDesc *will confirm that there exists an equivalent rewriting for q_1 using $\{v_1, v_2\}$.*

5.6.3 View descriptors from a QSS

We present in this section a bottom-up algorithm (findDescSupp) that runs on a QSS and a multi-token query q, computing view descriptors (w.r.t. q) for the expansions of the QSS. Our algorithm is sound and complete, running in polynomial time. Via Theorems 5.6.3 and 5.6.1, findDescSupp delivers a sound PTIME algorithm for support when the input queries are multi-token from XP . Moreover, via Theorems 5.6.3 and 5.6.2, it delivers a PTIME decision procedure for support when the input queries are multi-token from XP_{es}.

We will describe findDescSupp by separate subroutines, one for each of the four kinds of view descriptors: first-token descriptors in Section 5.6.3, last-token descriptors in Section 5.6.3, l-descriptors in Section 5.6.3 and m-descriptors in Section 5.6.3.

Since a QSS constructs views by putting together fragments, we construct our view descriptors via *fragment descriptors*, which represent equivalence classes for fragment expansions. Intuitively, fragment descriptors bookkeep in the bottom-up procedure certain partial details, on the expansions of fragment names, details that allow us to test *incrementally* the various conditions of testEquiv.

To better clarify our choices for fragment descriptors, let us first detail how the tests of testEquiv can be done in incremental manner.

Mapping and equivalence tests are naturally done bottom-up, one node at time, and this translates easily into procedures that run on the QSS and rely on descriptors. We already presented in Section 5.3 how one can test in this way the existence of containment or equivalence with q or parts thereof. We will handle the tests of lines 7 and 12 in testEquiv similarly, by descriptors which record mapping / equivalence details.

For line 9, the non-existence of a containment mapping between linear paths needs a slightly different approach. One can test incrementally if a linear path l_1 is contained in a linear path l_2 as follows:

- test if the last token of l_2 maps in the last token of l_1, such that $\text{OUT}(l_1)$ is the image of $\text{OUT}(l_2)$. Let k denote the start rank (the upmost node) of this mapping image.
- bottom-up, for each intermediary token t of l_2, map t in the *lowest possible*[1] available (i.e. above k) part of l_1. If no such mapping exists, we can conclude the non-existence of a containment mapping from l_2 in l_1. At each step, bookkeep in k the start rank of that image of t in l_1.
- finally, if the previous pass did not yield a negative answer already, a containment mapping of l_2 in l_1 does not exist if and only if the first token of l_2 cannot be mapped in l_1 s.t. (i) $\text{ROOT}(l_1)$ is the image of $\text{ROOT}(l_2)$, and (ii) the image of this first token of l_2 is above the current rank k.

The approach above advances one token at a time, and not one node at a time (which would have fitted nicely with how views are built in a QSS). This is because in order to test the *non-existence* of a certain mapping we need to check that all possible partial mappings fail sooner or later to go through to a full containment mapping (for line 9), root-mapping (for line 13), respectively output-mapping (for line 14). And the only way to ensure that no mapping opportunity is prematurely discarded is to settle on a mapping image in a descriptor, the lowest (resp. highest) possible one, only when a token is complete (i.e., it has an incoming //-edge).

A similar incremental approach, advancing one token at a time, can be used for the tests in lines 13 and 14, as we are dealing again with linear patterns. More

[1]Since we handle one token at a time, choosing the lowest available mapping image preserves all the opportunities to go through to a full containment mapping.

precisely, a bottom-up approach as above can be used in the case of MB($suffix_j$) and, symmetrically, a top-down one can be used in the case of MB($prefix_j$).

We are now ready to detail how view descriptors are computed in the algorithm findDescSupp. We start by assuming that all *equiv* and *map* descriptors are pre-computed for the boolean fragment names (as described in Section 5.3). In the same style, we compute *containment* and *equivalence descriptors* for *unary fragment names* (i.e. fragment names with an output mark). More precisely, a descriptor $contain(f, n)$, for $n \in \text{MBN}(q)$, (resp. $equiv(f, n)$) denotes that some expansion v_f contains (resp. is equivalent to) the suffix of q rooted at the main branch node n. Other types of fragment descriptors will also be introduced.

Computing first-token descriptors

For this part, we will use *prefix descriptors* for fragment names:

Definition 5.6.1. Syntax: *For a unary fragment name f, a* prefix descriptor *is a tuple pref(f, p, k), for k being a rank in the range* 1 *to* $|\text{MB}(ft)|$ *and p denoting any pattern that can be obtained from ft by keeping (a) a substring of the main branch, starting at the rank k, and (b) eventually, one predicate on that substring.*

Semantics: *There exists an expansion v_f s.t. (a) v_f has a containment mapping in the subtree of q rooted at the ft node of rank k, and (b) v_f has a first token which is of the form p plus additional predicates, if any (they are ignored in the descriptor).*

Step 1 **of** findDescSupp(q, $\mathcal{P}$)**.** Iterate the following steps:

1. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,

   add a prefix descriptor $\mathbf{pref}(\mathbf{f}, \mathbf{l}, \mathbf{k})$ for each rank k, $1 \leq k \leq |\text{MB}(ft)|$, for which we can infer that v_f contains the pattern $ft(k)$, by the following tests:

   - $node_{ft}(k)$ has label l,
   - for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_{ft}(k)$,

- for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_{ft}(k)$
- there exists a containment descriptor $contain(g, n)$ for n being any main branch node of rank $k' > k$ in q.

Moreover, if for a /-predicate (resp. //-predicate) P on $node_{ft}(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add the descriptor $\mathbf{pref}(\mathbf{f}, \mathbf{l}[\mathbf{P}], \mathbf{k})$.

2. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,

   given a prefix descriptor $pref(g, p', k')$, add a prefix descriptor $\mathbf{pref}(\mathbf{f}, \mathbf{l/p'}, \mathbf{k})$, for $k = k' - 1$, if we can infer that v_f contains the pattern $ft(k)$, by the following tests:

   - $node_{ft}(k)$ has label l,
   - for each fragment name c_i there exists a descriptor $map(c_i, n)$, for n being a /-child of $node_{ft}(k)$,
   - for each fragment name d_j there exists a descriptor $map(d_j, n)$, for n being a descendant of $node_{ft}(k)$

   Moreover, if for a /-predicate (resp. //-predicate) P on $node_{ft}(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{pref}(\mathbf{f}, \mathbf{l}[\mathbf{P}]/\mathrm{MB}(p'), \mathbf{k})$.

3. Whenever a descriptor *pref(f,p,1)* is obtained, for $f = S$, add $\mathbf{ft}(\mathbf{S}, \mathbf{p})$ to the set of view descriptors.

Computing last-token descriptors

We use for this part two kinds of fragment descriptors: *suffix descriptors* and *full-suffix descriptors.*

Definition 5.6.2. Syntax: *For a unary fragment name f, a* suffix descriptor *is a tuple suff(f, p), for p denoting any pattern that can be obtained from lt by keeping (a) a suffix of the main branch, and (b) eventually, one predicate on that suffix.*

Semantics: *This descriptor says that (a) v_f is a single-token query, of the form p plus maybe other predicates (ignored by the descriptor), and (b) v_f contains the subtree of lt rooted at the main branch node of rank $|\mathrm{MB}(lt)| - |\mathrm{MB}(p)| + 1$.*

Definition 5.6.3. Syntax: *For a unary fragment name f, a* full-suffix descriptor *is a tuple fsuff(f, p, k), for k denoting a rank in q, and p being a pattern as defined in Definition 5.6.2 above.*

Semantics: *There exists an expansion v_f s.t. (a) v_f has a last token of the form p plus other predicates (if any), and (b) v_f maps in the subtree of q rooted at the main branch node of rank k.*

Step 2 of findDescSupp(q, $\mathcal{P}$): summary.

We compute *suffix descriptors* similarly to the prefix ones. From them, *full-suffix descriptors* are then computed bottom-up, by simple containment mapping checks. If a descriptor *fsuff(f, p, 1)* is obtained, for $f = S$, we add $\mathbf{lt}(\mathbf{S}, \mathbf{p})$ to the set of view descriptors. We give below the detailed explanation of the algorithm.

For any rank k in p, by $cut(p, k)$ we denote the prefix of p having k main branch nodes.

Step 2 of findDescSupp(q, $\mathcal{P}$):

A. We compute *suffix descriptors* by iterating the following steps:

1. For $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,

   add a descriptor $\mathbf{suff}(\mathbf{f}, \mathbf{l})$ if we can infer that v_f contains the subtree of lt rooted at $\mathrm{OUT}(lt)$.

   if for a /-predicate (resp. //-predicate) P on $\mathrm{OUT}(lt)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{suff}(\mathbf{f}, \mathbf{l}[\mathbf{P}])$.

2. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,

   given a descriptor $suff(g, p')$, add a descriptor $\mathbf{suff}(\mathbf{f}, \mathbf{l}/\mathbf{p'})$ if, for $k = |\mathrm{MB}(q)| - |p'|$, we can infer that v_f contains the pattern $lt(k)$.

   if for a /-predicate (resp. //-predicate) P on $node_{lt}(k)$ we have a descriptor $equiv(c_i, root_P)$ (resp. $equiv(d_j, root_P)$), add also $\mathbf{suff}(\mathbf{f}, \mathbf{l}[\mathbf{P}]/\mathrm{MB}(p'))$.

B. Compute *full-suffix descriptors* by iterating the following:

1. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,

   given a suffix descriptor $suff(g, p)$, add a full suffix descriptor $\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k})$ for each rank $k < |\mathrm{MB}(q)| - |\mathrm{MB}(p)|$ s.t. we can infer that v_f contains the pattern $q(k)$.

2. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]/g(X)$,

   given a full-suffix descriptor *fsuff*(g, p, k'), add a descriptor $\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k})$, for $k = k' - 1$, if we can infer that v_f contains the pattern $q(k)$.

3. For $f(X) \rightarrow l[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]//g(X)$,

   given a full-suffix descriptor *fsuff*(g, p, k'), add a descriptor $\mathbf{fsuff}(\mathbf{f}, \mathbf{p}, \mathbf{k})$ for each rank $k < k'$ s.t. we can infer that v_f contains the pattern $q(k)$.

4. Whenever a descriptor *fsuff*$(f, p, 1)$ is obtained, for $f = S$, add $\mathbf{lt}(\mathbf{S}, \mathbf{p})$ to the set of view descriptors.

Computing l-descriptors

We have seen in Section 5.6.3 an incremental procedure that tests the non-existence of a containment mapping bottom-up, one token at a time. In order to run a similar test directly on the QSS (whose expansions are revealed one node at a time), we need some additional bookkeeping, allowing us to chose mapping images one token at a time. For this, we record at each step in the bottom-up process the following: (i) the current first token of v_f, and (ii) the *lowest possible* mapping image for only the rest of v_f (except its first token). This allows us to settle on the lowest possible mapping (in a descriptor) only when the token is complete (i.e., we have its incoming edge and it is a //-edge). To this end, we use *partial l-descriptors*.

Definition 5.6.4. Syntax: *For a unary fragment name f, a* partial l-descriptor *is a tuple $pl[f, k_1, (k_2, p)]$, where k_1 is a rank in q, k_2 is a rank in l_q and p is any substring of l_q.*

Semantics: *There exists an expansion v_f s.t. (a) v_f contains the subtree of q rooted at the main branch node of rank k_1, (b) the main branch of the first token of v_f is p, and (c) k_2 is the start (upmost rank) of the* lowest possible output-mapping image *of the rest of the main branch of v_f (i.e., except p) into l_q. By convention, this rank is $|l_q| + 1$ when v_f has only one token (the one described by p) and is 0 when there is no such mapping.*

Step 3 of findDescSupp(q, $\mathcal{P}$). Iterate the following steps:

1. For rules $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,

   if we can infer that v_f contains the subtree of q rooted at $\mathrm{OUT}(q)$, add a descriptor $\mathbf{pl}[\mathbf{f}, |\mathrm{MB}(q)|, (|\mathbf{l_q}| + \mathbf{1}, \mathbf{l})]$

2. For $f(X) \rightarrow l[\ldots]/g(X)$, given a descriptor $pl[g, k_1', (k_2', p')]$, if we can infer that v_f contains the pattern $q(k_1' - 1)$:

   - if f is not the start fragment name, add the descriptor $\mathbf{pl}[\mathbf{f}, \mathbf{k_1' - 1}, (\mathbf{k_2'}, \mathbf{l/p'})]$.
   - otherwise, if there is no mapping of l/p' into l_q whose image starts at $\mathrm{ROOT}(l_q)$ and ends above k_2', add the descriptor $\mathbf{l}(\mathbf{S})$ to the set of view descriptors.

3. For $f(X) \rightarrow l[\ldots]//g(X)$ and a descriptor $pl[g, k_1', (k_2', p')]$, for each rank k_1, $1 \leq k_1 < k_1'$, s.t. we can infer that v_f contains the pattern $q(k_1)$:

   - if f is not the start fragment name, find the lowest rank k_2, s.t. p' has a mapping into l_q whose image starts at k_2 and ends *above* k_2', where if $k_2' = |l_q| + 1$ above means at $k_2' - 1$; if no such value exists, set k_2 to 0. Output the descriptor $\mathbf{pl}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})]$.
   - otherwise, if there is no mapping of $l//p'$ into l_q whose image starts at $\mathrm{ROOT}(l_q)$ and ends above k_2', add the descriptor $\mathbf{l}(\mathbf{S})$ to the set of view descriptors.

Computing m-descriptors

For this part, we need to check that some view v_j can be seen as being of the form $\mathit{prefix}_j//m'//\mathit{suffix}_j$, s. t. $m' \equiv m$ and

- prefix_j root-maps into ft but $\mathrm{MB}(\mathit{prefix}_j)$ cannot root-map into $\mathrm{MB}(ft)_m$,
- suffix_j output-maps into lt, but $\mathrm{MB}(\mathit{suffix}_j)$ cannot output-map into $\mathrm{MB}(lt)_m$.

Each of these aspects of an expansion is captured by a different type of fragment descriptor. They enable us to output a view descriptor $\mathbf{m}(\mathbf{S})$ when a rule $f(X) \rightarrow l[\ldots]//g(X)$ is available and when (via fragment descriptors) we have that:

- g has expansions that give us the part $m'//\mathit{suffix}_j$,
- there exist views generated via that rule and g, s.t. the part above g's expansion (in other words, the view obtained by expanding g in the empty pattern) has the properties for prefix_j.

We can use separate subroutines for each of these two items, and then the overall step above will combine their individual results.

For the suffix_j part, we use *below m-descriptors*:

Definition 5.6.5. Syntax: *For a unary fragment name f, a* below m-descriptors *is a tuple $bm[f, k_1, (k_2, p)]$, where k_1 and k_2 denote ranks in q, and p denotes any substring of* $\mathrm{MB}(q)$.

Semantics: *There exists an expansion v_f s.t. (a) v_f contains the subtree of ft rooted at the node of rank k_1, (b) p is the main branch of the first token of v_f, and (c) k_2 is the start of the* lowest possible output-mapping image *of the main branch of the rest of v_f (besides p) into* $\mathrm{MB}(lt)_m$*; by convention, k_2 is $|\mathrm{MB}(q)| + 1$ when v_f has only one token and is* 0 *when there is no such mapping.*

Then, for the m part, we use *partial m-descriptors*:

Definition 5.6.6. Syntax: *For a unary fragment name f, a* partial m-descriptor *is a tuple $pm(f, k)$, where k is a number in the range* 1 *to* $|\mathrm{MB}(m)|$*, indicating a suffix of m.*

Semantics: *This descriptor says that (a) v_f is of the form $m'//\mathit{suffix}_j$, s.t. m' is equivalent with m's suffix having k main branch nodes, and (b) suffix_j has*

the properties described above.

For the $prefix_j$ part, we use *above m-descriptors*:

Definition 5.6.7. Syntax: *For a unary fragment name f, an* above m-descriptor *is a tuple $am[f, k_1, (k_2, p)]$, where k_1, k_2 denote ranks in q and p is any substring of* $\mathrm{MB}(q)$.

Semantics when p is empty (denoted hereafter '–'): *there exists an expansion v of the QSS s.t.*

(a) v is of the form $rest//v_f$, for v_f being an expansion of f,

(b) $rest$ root-maps into ft such that its image ends at the rank k_1, and

(c) the end (bottommost node) of the highest possible root-mapping *image of* $\mathrm{MB}(rest)$ *into* $\mathrm{MB}(ft)_m$ *is k_2; if no such mapping exists, by convention k_2 is* $|\mathrm{MB}(ft)_m| + 1$.

Semantics when $p \neq$ '–': *there exists an expansion v of the QSS s.t.*

(a) v is of the form $rest//p'/v_f$, for $p = \mathrm{MB}(p')$,

(b) $rest//p'$ root-maps into ft such that the image of p' ends at the rank k_1, and

(c) the end (bottommost node) of the highest possible root-mapping *image of* $\mathrm{MB}(rest)$ *into* $\mathrm{MB}(ft)_m$ *is k_2; by convention, if no such mapping exists, k_2 is* $|\mathrm{MB}(ft)_m| + 1$*; when $rest$ is empty k_2 is* 0.

Given a rule $f(X) \to l[\ldots]/g(X)$ or $f(X) \to l[\ldots]//g(X)$, we will use an am-descriptor for f to infer one for g.

Step 4 of findDescSupp(q, $\mathcal{P}$): summary.

Below m-descriptors are computed by a similar approach (one token at time) as the one used for partial l-descriptors. The *above m-descriptors* are obtained similarly, but in top-down manner. Starting from below-m descriptors, the *partial m-descriptors* are computed bottom-up, by simple equivalence checks.

If for some fragment name g we computed both an above m-descriptor $am[g, k_1, (|\text{MB}(ft)_m| + 1, -)]$ and a partial m-descriptor $pm(g, |\text{MB}(m)|)$, we can add a descriptor $\mathbf{m}(\mathbf{S})$ to the set of view descriptors.

Here is the detailed presentation of how m-descriptors are computed.

Step 4 **of** findDescSupp(q, $\mathcal{P}$)**.** Apply the the following steps:

A. Compute *below m-descriptors* by iterating the following steps:

1. For rules $f(X) \rightarrow l(X)[c_1(), \ldots c_n(), .//d_1(), \ldots, .//d_m()]$,

   if we can infer that v_f contains the subtree of q rooted at $\text{OUT}(q)$, add a descriptor $\mathbf{bm}[\mathbf{f}, |\text{MB}(q)|, (|\text{MB}(q)| + \mathbf{1}, \mathbf{l})]$.

2. For $f(X) \rightarrow l[\ldots]/g(X)$ and a descriptor $bm[g, k_1', (k_2', p')]$, if we can infer that v_f contains the pattern $lt(k_1'-1)$, add the descriptor $\mathbf{bm}[\mathbf{f}, \mathbf{k_1'} - \mathbf{1}, (\mathbf{k_2'}, \mathbf{l}/\mathbf{p'})]$.

3. For $f(X) \rightarrow l[\ldots]//g(X)$, a descriptor $bm[g, k_1', (k_2', p')]$, for each rank k_1, $k_1 < k_1'$, s.t. we can infer that v_f contains the pattern $lt(k_1)$,

   find the lowest possible rank k_2 in $\text{MB}(lt)_m$ s.t. the token p' has a mapping into $\text{MB}(lt)_m$ starting at k_2 and ending *above* k_2', where if $k_2' = |\text{MB}(q)| + 1$ above means at $k_2' - 1$; if no such rank is found, set k_2 to 0.

   add a below m-descriptor $\mathbf{bm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})]$.

B. Compute *partial m-descriptors* by iterating the following steps:

1. For $f(X) \rightarrow l[\ldots]//g(X)$, given a below m-descriptor $bm[g, k_1', (k_2', p')]$ s.t. (i) k_2' is already 0, or (ii) p' cannot be mapped anywhere above k_2' in $\text{MB}(lt)_m$

   if we can infer that v_f' is equivalent with m's suffix of size 1, add the partial m-descriptor $\mathbf{pm}(\mathbf{f}, \mathbf{1})$.

2. For either $f(X) \rightarrow l[\ldots]/g(X)$ or $f(X) \rightarrow l[\ldots]//g(X)$

   given a partial m-descriptor $pm(g, n')$: if we can infer that the query $cut(v_f, n'+1)$ is equivalent with the suffix of m of size $n = n' + 1$, add a descriptor $\mathbf{pm}(\mathbf{f}, \mathbf{n})$.

C. Compute *above m-descriptors* by iterating the following steps:

1. From start fragment names f and either rules

   $f(X) \rightarrow l[\ldots]/g(X)$ or $f(X) \rightarrow l[\ldots]//g(X)$

   if we can infer that v'_f root-maps into ft, add a descriptor $\mathbf{am}[\mathbf{g}, \mathbf{1}, (\mathbf{0}, \mathbf{l})]$ (respectively $\mathbf{am}[\mathbf{g}, \mathbf{1}, (\mathbf{1}, -)]$).

2. For $f(X) \rightarrow l[\ldots]/g(X)$ and a descriptor $am[f, k'_1, (k'_2, p')]$:

   - *if p' is not '-':* if we can infer that v'_f root-maps in $ft(k'_1 + 1)$, add a descriptor $\mathbf{am}[\mathbf{g}, \mathbf{k'_1 + 1}, (\mathbf{k'_2}, \mathbf{p'/l})]$
   - *if p' is '-':* for each rank k_1, $k_1 > k'_1$, s.t. we can infer that v'_f root-maps in the pattern $ft(k_1)$,add a descriptor $\mathbf{am}[\mathbf{g}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l})]$.

3. For $f(X) \rightarrow l[\ldots]//g(X)$, a descriptor $am[f, k'_1, (k'_2, p')]$:

   - *if p' is not '-':* if we can infer that v'_f root-maps in the pattern $ft(k'_1+1)$, add a descriptor

     $$\mathbf{am}[\mathbf{g}, \mathbf{k'_1 + 1}, (\mathbf{k_2}, -)]$$

     for the highest rank k_2, $k'_2 < k_2 \leq |\mathrm{MB}(ft)_m|$, s.t. p'/l has a mapping into $\mathrm{MB}(ft)_m$ starting *below* k'_2, where if $k'_2 = 0$ below means rank 1, and ending at k_2; if no such mapping exists set k_2 to $|\mathrm{MB}(ft)_m| + 1$.
   - *if p' is '-':* for each rank k_1, $k_1 > k'_1$, s.t. v'_f root-maps in the pattern $ft(k_1)$, add

     $$\mathbf{am}[\mathbf{g}, \mathbf{k_1}, (\mathbf{k_2}, -)]$$

     for the highest rank k_2, $k'_2 < k_2 \leq |\mathrm{MB}(ft)_m|$, s.t. the token l has a mapping into $\mathrm{MB}(ft)_m$ at rank k_2; if no such mapping exists set k_2 to $|\mathrm{MB}(ft)_m| + 1$.

D. Finally, for a fragment name g, given both

- a partial m-descriptor $pm(g, |\mathrm{MB}(m)|)$,
- an above m-descriptor $am[g, k_1, (|\mathrm{MB}(ft)_m| + 1, -)]$,

add a descriptor $\mathbf{m}(\mathbf{S})$ to the set of view descriptors.

We can now prove the following:

Theorem 5.6.4. *Given a QSS $\mathcal{P}$ and a query q, algorithm* findDescSupp *is sound and complete for computing the descriptors for $\mathcal{P}$'s expansions.* findDescSupp *runs in polynomial time in the size of the query and of the QSS.*

Proof: [Sketch]

First-token descriptors. Step 1 of findDescSupp computes, bottom up, all suffixes of a prefix plus, eventually, one predicate. It stores them in the second field of the **pref** descriptor. When the position $k = 1$ is reached, it means that a main branch of a first token, plus maybe a predicate, has been computed. This justifies the inference of an **ft** descriptor that has the same p pattern as the **pref**(f, p, 1) descriptor. We infer all such descriptors because we explore bottom-up all paths that could represent the first-token in an expansion of the QSS.

Last-token descriptors. This case is symmetrical to the first-token descriptors.

l-descriptors. Partial l-descriptors are computed starting from the base case of rules that have no tree fragment names (Step 1). Then it records bottom up patterns and ranks in q and l_q, inferring partial l-descriptors that satisfy Definition 5.6.4. It infers all such descriptors because, intuitively, a partial l-descriptor for smaller ranks in q and l_q exists only if there are partial l-descriptors for higher ranks, i.e., corresponding to "lower" fragments of the main branches of the tree patterns. And, by the same reasoning, if no mapping can be inferred while going "up" in the pattern (case in which an **l** descriptor is inferred), it guarantees the non-existence of a containment mapping.

m-descriptors. Below m-descriptors are computed in a very similar way to partial l-descriptors. And, using similar arguments, it can be shown that the algorithm computes all below m-descriptors. The computation of partial m-descriptors follows exactly the conditions in their definition. Above m-descriptors are also computed as a bottom-up evaluation, checking the conditions from Definition 5.6.7. Finally, having a partial m-descriptor $pm(g, |\mathrm{MB}(m)|)$ and an above m-descriptor $am[g, k_1, (|\mathrm{MB}(ft)_m| + 1, -)]$ justifies the introduction of an **m** de-

scriptor, as it guarantees that some views generated by the QSS satisfy the conditions from lines 11-14 in Algorithm testEquiv.

It can easily be verified that the number of descriptors is polynomial, as they are defined using positions, subpatterns or patterns constructed in PTIME from the query and the specification. The computation of each descriptor is PTIME, as it amounts to simple tests on polynomial size patterns. Hence the computation of all descriptors is done in PTIME. ■

By Theorems 5.6.4, 5.6.3 and 5.6.1, for a multi-token XP query q and QSS $\mathcal{P}$, given the descriptor set $\mathcal{D} :=$ findDescSupp$(q, \mathcal{P})$, q is supported by $\mathcal{P}$ if testEquivDesc(q, $\mathcal{D}$) outputs true.

Moreover, by Theorem 5.6.2, if q is from XP_{es}, it is supported by $\mathcal{P}$ (considering for now only rewrite plans that intersect some of the views) if and only if testEquivDesc(q, $\mathcal{D}$) outputs true. We generalize these two observations to support in $XP^{\cap}$ in the next section.

5.6.4 Support with compensated views

We consider in this section general $XP^{\cap}$ rewrite plans for support that, before performing the intersection step, might compensate (some of) the views. We show that support in this new setting can be reduced to support by rewrite plans which only intersect expansions of a QSS. This allows us to reuse the PTIME algorithms given in Section 5.6 (testEquivDesc and findDescSupp) and to find strictly more rewritings, namely those that would not be feasible without compensation. Thus we obtain a sound algorithm for support on XP multi-token queries in the rewrite language $XP^{\cap}$. This algorithm becomes complete when the input query is from XP_{es}.

Our reduction relies on the same QSS transformation, $comp(\mathcal{P}, q)$, used in Section 5.4.1, which builds expansions with compensation.

EXAMPLE 5.6.4. *Suppose that the QSS of the source in Example 5.2.1 is modified to return the guided trips themselves instead of the museums of those trips, by changing the third rule into rule R_3:*

$$(R_3): \quad f_1(X) \rightarrow trip(X)[guide].$$

and obtaining a new QSS $\mathcal{P}_2$. Then, one of the expansions of $\mathcal{P}_2$ is:

$$v_3\text{: } doc(T)//vacation//trip/trip[guide]$$

A query plan that rewrites q_2 using compensated views is

$$doc(v_3)/v_3/trip/museum \cap doc(v_2)/v_2/museum.$$

We can infer this rewriting by compensating R_3 with a navigation to a museum child, which leads to a QSS identical to $\mathcal{P}$.

We can prove the following:

Theorem 5.6.5. *Given a QSS $\mathcal{P}$ and a multi-token XP query q, let $\mathcal{D} :=$ $\mathsf{findDescSupp}(q, comp(\mathcal{P}, q))$.*

1. *Algorithm* $\mathsf{testEquivDesc}(q, \mathcal{D})$ *is sound for support in $XP^{\cap}$, i.e., q is supported by $\mathcal{P}$ in $XP^{\cap}$ if* $\mathsf{testEquivDesc}(q, \mathcal{D})$ *outputs true.*

2. *Algorithm* $\mathsf{testEquivDesc}(q, \mathcal{D})$ *is also complete if q belongs to XP_{es}, i.e., q is supported by $\mathcal{P}$ in $XP^{\cap}$ if and only if* $\mathsf{testEquivDesc}(q, \mathcal{D})$ *outputs true.*

Proof: [Sketch]

1. Soundness follows from the observation that follows after Theorem 5.6.4 and from the fact that $comp(\mathcal{P}, q))$ generates compensated views.

2. Follows from Theorem 5.6.2 and from the fact that $comp(\mathcal{P}, q))$ generates views having all the compensation that may be used by a mapping from q into an interleaving of views.

■

5.7 Single-token queries

We consider in this section the remaining sub-fragment of XP_{es}, namely single-token queries. We show that id-based support becomes NP-hard (Theorem

5.7.1). Contrast this with both id-support for queries that have at least one //-edge in the main branch, and the rewriting problem for single-token XP_{es} queries under an explicit set of views, for which PTIME decision procedures exist.

Theorem 5.7.1. *For an XP_{es} single-token query q and a QSS $\mathcal{P}$, deciding if q is supported by $\mathcal{P}$ in $XP^{\cap}$ is NP-hard.*

Proof: We use a reduction from the minimum set-cover problem [GJ79]. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \ldots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \ldots S_m\}$ denoting the sets s.t. $S_i \subset \mathcal{U}$ for each S_i. We want to know whether there exists a subset S' of S, of size at most k, that can cover $\mathcal{U}$ (i.e. each element of $\mathcal{U}$ belongs to at least one set of S').

The reduction takes as input the set $\mathcal{U}$ and $\mathcal{S}$ (size $|\mathcal{S}| \times |\mathcal{S}|$) and the value k (size $lg(k)$).

Let p be the biggest exponent s.t. $2^p \leq k$ and let $b_p b_{p-1} \ldots b_0$ be the binary representation of k.

We build the following instance of the support problem. We define the QSS as follows:

- the tree fragments

$$F = \{S, set, f, g, f_s, f_p^p, f_p^{p-1}, \ldots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \ldots, f_{p-1}^0, \ldots, f_1^1, f_1^0, f_0^0\}$$

- the alphabet $\Sigma = \{root, out, a, b, e_1, \ldots, e_m\}$
- S is the start fragment
- the set of productions P defined as follows:

  1. the start productions

$$S(X) \rightarrow root/f(X)$$

  and

$$S(X) \rightarrow root//g(X)$$

2. $f(X) \rightarrow a/a/a[Q]/a/out(X)$

   Where Q denotes the pattern obtained as follows: let L_i, $i = \overline{0,p}$, denote the pattern formed by $i+2$ nodes labeled b followed by the predicate $[e_1, \ldots, e_n]$. For example, $L_0 = b[b[e_1, \ldots, e_n]]$.

   Then, we define Q as

$$Q = b[L_p][L_{p-1}] \ldots [L_1][L_0].$$

3. $g(X) \rightarrow a[b[//e_1, \ldots, //e_n]]/a[f_s]/a//out(X)$
4. $f_s \rightarrow b[f_p^p, f_{p-1}^{p-1} \ldots, f_1^1, f_0^0]$
5. $\forall i = \overline{0,p}$ s.t. $b_i = 1$, we have the productions

$$f_i^k \rightarrow b[f_i^{k-1}][f_i^{k-1}] \forall k = \overline{0,i}$$

$$f_i^0 \rightarrow b[set]$$

6. $\forall i = \overline{0,p}$ s.t. $b_i = 0$, we have the production

$$f_i^i \rightarrow ()$$

7. $\forall S_j = \{e_{l_1}, \ldots, e_{l_j}\} \in S$ we have the production

$$set \rightarrow b[e_{l_1}, \ldots, e_{l_j}]$$

8. finally we have the production

$$set \rightarrow ()$$

This QSS produces two families of views. The first one, by the fragment name f, contains only one member (v_f), which has a main branch $root/a/a/a/a/out$, and a Q predicate on the third a-node.

The views from the second family of views, by the fragment name g, have the main branch $root//a/a/a//out$, the predicate $b[//e_1, //e_2, \ldots, //e_n]$ on the first a-node and a predicate produced by f_s on the second a-node. f_s produces branches of length at most $p+2$ followed by elements $e_1, \ldots, e_n$.

Let now q be the single-token XP_{es} query

$$q = root/a/a[b[//e_1, //e_2, \ldots, //e_n]]/a[Q]/a/out.$$

It is easy to see that all the views generated by the QSS contain q: v_f maps obviously in q and all the other views have a containment mapping into q since, by construction, any pattern produced by f_s can be mapped into Q.

We now consider if q is supported, which amounts here to testing if the intersection of all the views is contained in q. Note that q will contain any interleaving which, for at least one view v_g, collapses the third a-node of v_g with the fourth a-node of v_f. This is because such an interleaving would be of the form

$$root/a/a[b[//e_1, //e_2, \ldots, //e_n]][\ldots]/a[Q][Q'][\ldots]/a/out$$

where Q' is produced by f_s and is actually redundant (can be minimized away).

So the only interleavings that remain to consider are those in which all the third a-nodes of v_g views are collapsed with the third a-node of v_f. These interleavings are of the form

$$root/a[b[//e_1, //e_2, \ldots, //e_n]]/a[Q'][\ldots]/a[Q]/a/out$$

We can see now that q contains these interleavings if and only if among the predicates Q' produced by f_s there is one into which the pattern $b[//e_1, \ldots, //e_n]$ can map. But this is possible if and only if all the elements $e_1, \ldots, e_n$ are present in Q', and this happens if and only if there exists a cover of maximal size k for the set $\mathcal{U}$. ■

The surprising dichotomy between support for single-token and multi-token extended skeletons is rooted in their differences on the respective tests for equivalence with an intersection of views.

First, for the single-token case, it is easy to see that support can hold only if the main branch of some view is equivalent to q's main branch. Otherwise, one could easily exhibit interleavings that do have //-edges in their main branch, hence cannot be contained in q. With this, building interleavings amounts basically to deciding where to collapse main branch nodes from the various views on a linear

path with /-edges only. Intuitively, it is now less a matter of how to order main branch nodes of the views, and more of choosing for each node a coalescing option among the few available. By consequence, a candidate interleaving i (i.e., one that is equivalent to q and contains all other interleavings) might combine (put under the same main branch node) predicates coming from different views *at all levels of the main branch*. When q has several tokens, this is true only for the candidate's first and last tokens (built by combining in the only possible way the first and last tokens of the views), while the section in between has to be entirely present (isomorphic modulo minimization) in some view. In the following, we describe a sound, tractable procedure for support when the user query q is a single-token XP_{es} query. We show how it can be extended to an exponential-time sound and complete algorithm for this problem.

For presentation simplicity, we first limit the discussion to rewrite plans that are intersections of views (no compensation before the intersection step). We start by considering how one can check the existence of a rewriting using a finite set of explicitly listed views.

For the to-be-rewritten query q and a predicate P in q, for a view v let N_v^P denote the set of main branch nodes of v having a predicate P' s.t. (i) the pattern of P' is equivalent to some subtree of q, and (ii) the pattern q_P contains the pattern $v_{P'}$.

We are now ready to formulate a sound, tractable algorithm which, for a set of explicitly listed views $\mathcal{V}$, tests if an equivalent rewriting exists for a single-token query q.

The tests we present ensure that the opposite containment mapping holds, i.e., that q contains each possible interleaving of the views. Section 5.7.1 details how one can verify these properties even when abstracting away from the view definitions, by using view descriptors, and in Section 5.7.2 we show how descriptors can be inferred when views are defined as the expansions of a QSS.

For a view v_j verifying the conditions of lines 7-9, we say that v_j *contributes* P in the intersection. We can prove the following:

Algorithm 3 testEquivSingle($\mathcal{V}, q$)

```
1: begin
2: for each predicate P in q and each view v_j ∈ V
3:    compute the set of nodes N^P_{v_j}
4: if there exists a view v_i ∈ V s.t. MB(v_i) ≡ MB(q) then
5:    if for all predicates P in q
6:       there exists a view v_j ∈ V s.t.
7:          for all containment mappings ψ from MB(v_j) into MB(q)
8:                the node n_P is an image under ψ
9:                the node n ∈ v_j for which ψ(n) = n_P is in N^P_{v_j}
10:   then output true
11: end
```

Theorem 5.7.2. *For an XP single-token query q and a set of XP views $\mathcal{V}$ containing q,* testEquivSingle *is a sound PTIME procedure for deciding if $q \equiv \cap\mathcal{V}$.*

Proof: [Sketch] When q has only /-edges in the main branch, among the views of $\mathcal{V}$ there must be at least one having the same main branch as q. With this, the intersection yields only interleavings having as main branch that linear path with /-edges only. The only remaining issue is whether all of q's predicates will be "present" at each interleaving, thus enabling a containment mapping from q. This follows from conditions (lines 7-9) which imply that, in any interleaving i, a predicate into which P can map will be present at i's main branch node of rank r_P.

Testing conditions (lines 7-9) for each predicate P in q can be easily translated into a containment mapping test. ■

5.7.1 Descriptors

Again, the key idea for checking support when views are defined by a QSS is that, in order to test the conditions of the algorithm testEquivSingle, one does not need the detailed definitions of the views but only some particular details on them. This idea will be exploited by our *view descriptors*. More precisely, assuming we

are dealing with expansions of a QSS $\mathcal{P}$ with start fragment name S,

For the condition of line 4. For a view v containing q, a view descriptor $\mathbf{mb}(\mathbf{S})$ will indicate that the pattern representing the main branch of v is equivalent with $\mathrm{MB}(q)$.

For the conditions of lines 7-9. A direct but expensive (complexity-wise) solution for this part would be specify in a descriptor the set of predicates P the view contributes, according to lines $7-9$. But it suffices instead to consider predicates individually. By a descriptor $\mathbf{pred}(\mathbf{S}, \mathbf{P})$ we denote a view v which contains q and contributes predicate P. The fact that a certain view may contribute several predicates is irrelevant, as it is enough to know that there exist covering views for each of them (not necessarily distinct).

Note that descriptors partition the set of views into equivalence classes with respect to the tests of testEquivSingle: two views having different definitions but yielding the same descriptors will be equally useful for these tests.

Now, we can easily rephrase testEquivSingle into an algorithm that runs directly on the set of view descriptors D, instead of the explicit views $\mathcal{V}$ to which they correspond.

Algorithm 4 testEquivSingleDesc(D, q)

1: **if** there exists a descriptor $\mathbf{mb}(\mathbf{S}) \in D$ **then**
2: **if for all** predicates P in q
3: **there exists** a descriptor $\mathbf{pred}(\mathbf{S}, \mathbf{P}) \in D$
4: **then** output **true**
5: **end**

We can prove the following:

Theorem 5.7.3. *For a single-token query q, a finite set of views $\mathcal{V}$ containing q and their corresponding descriptors D,* testEquivSingle*(q, V) outputs* true *if and only if* testEquivSingleDesc*(q, D) does so.*

5.7.2 Descriptors from a QSS program

We present in this section a bottom-up algorithm that computes, for a QSS and a single-token query q, all the view descriptors for all the QSS expansions.

As before, we need fragment descriptors in order to perform incrementally the tests of **testSupp**. While this is straightforward for the test of line 4, the conditions of lines $7-9$ require special treatment. More precisely, given a view v_j and a query q, we rephrase this part into testing the non-existence of a containment mapping (with a twist) between $\mathrm{MB}(v_j)$ and $\mathrm{MB}(q)$ (i.e., linear patterns). Note that this is something we already know how to do from the multi-token case. For that, since there is always at least one containment mapping from $\mathrm{MB}(v_j)$ into $\mathrm{MB}(q)$, we will change some of the labels in $\mathrm{MB}(v_j)$ and $\mathrm{MB}(q)$ in order to test if there are mappings violating the conditions of lines $7-9$.

Assuming that the set of nodes $N_{v_j}^P$ is not empty, let l_P denote the label of n_P and let l'_P denote a new label derived from it. For this new label, let us consider the following variation to the usual definition of a mapping: *a node of label l'_P can map into a node of label l_P.* This variation will be called hereafter *P-mapping*. Based on this, containment P-mapping is then defined in the usual way. Using this new label, let $relabel(P, v_j)$ denote the pattern obtained from v_j by relabeling with l'_P the nodes that were labeled l_P in v_j and were not in the set $N_{v_j}^P$. Also, let q' denote the pattern obtained from q by relabeling the node n_P by l'_P.

It is now easy to see that testing the conditions of lines $7-9$ amount to testing the non-existence of a containment P-mapping from $\mathrm{MB}(relabel(P, v_j))$ into $\mathrm{MB}(q')$. And this can be done on an explicit view v_j by the same bottom-up approach, which advances one token at a time, described in Section 5.6.3.

For each predicate P, we describe a subroutine that finds the descriptors **pred**(**P**). The **mb**() one will be obtained as a side-effect of these subroutines. We assume that the *map*, *equiv*, *contain* descriptors are already pre-computed. Besides those, we will use only one kind of fragment descriptors, called *intermediary descriptors*.

Definition 5.7.1. Syntax: *An intermediary descriptor w.r.t. a fixed predicate P in q for a unary fragment name f is a tuple $interm[f, k_1, (k_2, p)]$, where k_1 and k_2*

are ranks in the main branch of q*, and* p *denotes any linear substring of* $\mathrm{MB}(q)$ *or* $\mathrm{MB}(q')$*.*

Semantics:

- v_f *contains the subtree of* q *rooted at the main branch node of rank* k_1*,*
- p *is the main branch of the first token of* $relabel(P, v_f)$*,*
- k_2 *is the start rank (i.e., the upmost node) of the* lowest possible output P-mapping image *of the rest of the main branch of* $relabel(P, v_f)$ *(besides* t*) into* $\mathrm{MB}(q')$*; by convention,* k_2 *is* $|\mathrm{MB}(q)| + 1$ *when* v_f *has only one token (the one described by* t*) and is* 0 *when there is no such P-mapping.*

We remind that for a tree pattern p and a node $n \in \mathrm{MBN}(p)$, by $\mathrm{SP}_p(n)$ we denote the subtree rooted at n in p.

Algorithm findDescriptors:

For each predicate P in q, repeat until fix-point:

1. for rules $f(X) \to l(X)[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()]$,

   if we can infer that v_f contains the subtree of q rooted at $\mathrm{OUT}(q)$, add a descriptor $\mathbf{interm}[\mathbf{f}, |\mathrm{MB}(q)|, (|\mathrm{MB}(q) + \mathbf{1}|, \mathbf{l})]$.

2. for rules $f(X) \to l[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()]/g(X)$,

   given a descriptor $interm[g, k_1', (k_2', p')]$, for $k_1 = k_1' - 1$, if we can infer that v_f contains the subtree $q(k_1)$, then

   (a) if $l \neq l_P$, add a descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2'}, \mathbf{l/p'})]$,

   (b) if $l = l_P$: add a descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2'}, \mathbf{l_P/p'}]$

   (i) if P is a /-predicate and there exists a descriptor $equiv(c_i, n)$ s.t.

   - the pattern q_P contains the pattern $l/xpath(\mathrm{SP}_q(n))$,
   - the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some /-child of $node_q(k_1)$,

   or (ii) if P is a //-predicate and there is a descriptor $equiv(d_j, n)$ s.t.

- the pattern q_P contains the pattern $l//xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some descendant of $node_q(k_1)$.

(c) otherwise, add the descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l'_P}/\mathbf{p'})]$.

If f is the start fragment name S, if $k'_1 - 1 = 1$ and if we can infer that v_f contains q,

(a) if $k'_2 = 0$ or the token l/p' does not have a P-mapping image in $\mathrm{MB}(q')$ starting at $\mathrm{ROOT}(q')$ and ending *above* k'_2, where if $k'_2 = |\mathrm{MB}(q)| + 1$ above means at $k'_2 - 1$, add a view descriptor $\mathbf{pred}(\mathbf{P})$

(b) moreover, if $|l/p'| = |\mathrm{MB}(q)|$ (which here means $\mathrm{MB}(v_f) \equiv \mathrm{MB}(q)$), add also the descriptor $\mathbf{mb}(\mathbf{S})$ to the set of view descriptors.

3. for $f(X) \rightarrow l[c_1(), \ldots c_n, .//d_1(), \ldots, .//d_m()]//g(X)$,

given a descriptor $interm[g, k'_1, (k'_2, p')]$, for each rank k_1, $1 \leq k_1 < k'_1$, s.t. we can infer that v_f contains $q(k_1)$,

find also the lowest possible rank k_2, $1 \leq k_2 < k'_2$, such that the token p' P-maps into $\mathrm{MB}(q')$ starting at k_2 and ending *above* k'_2, where if $k'_2 = |\mathrm{MB}(q)| + 1$ above means at $k'_2 - 1$; if no such rank is found, set k_2 to 0.

(a) if $l \neq l_P$, add a descriptors $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k_2}, \mathbf{l})]$,

(b) if $l = l_P$: add the descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l_P}]$

(i) if P is a /-predicate and there exists a descriptor $equiv(c_i, n)$ s.t.

- the pattern q_P contains the pattern $l/xpath(\mathrm{SP}_q(n))$,
- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some /-child of $node_q(k_1)$,

or (ii) if P is a //-predicate and there is a descriptor $equiv(d_j, n)$ s.t.

- the pattern q_P contains the pattern $l//xpath(\mathrm{SP}_q(n))$,

- the pattern $\mathrm{SP}_q(n)$ root-maps in the subtree of q rooted at some descendant of $node_q(k_1)$.

(c) otherwise, add the descriptor $\mathbf{interm}[\mathbf{f}, \mathbf{k_1}, (\mathbf{k'_2}, \mathbf{l'_P})]$.

When $f = S$, if the token l does not P-map into $\mathrm{MB}(q')$ starting at rank 1 and ending above k_2 (i.e., $k_2 = 0$ or $k_2 = 1$), add a descriptor $\mathbf{pred}(\mathbf{S}, \mathbf{P})$ to the set of view descriptors.

Theorem 5.7.4. *Given a QSS $\mathcal{P}$ and a single-token query q, algorithm* ***findDescriptors*** *is sound and complete for computing the descriptors for $\mathcal{P}$'s expansions. It runs in polynomial time in the size of the query and of the QSS.*

5.7.3 Decision procedure for support

We describe in this section a sound and complete algorithm for support on single-token XP_{es} queries which runs in exponential time in the size of the query. For this, we relax the definition of the N_v^P set as follows: for the to-be-rewritten query q and a predicate P in q, for a view v, N_v^P now denotes the set of main branch nodes of v having a predicate P' s.t. the pattern q_P contains the pattern $v_{P'}$.

With this adjustment, let decideSuppSingle denote the corresponding procedure. We can prove the following:

Theorem 5.7.5. *For an XP_{es} single-token query q and a set of XP views $\mathcal{V}$, in which each view contains q,* decideSuppSingle *is a sound and complete procedure for $q \equiv \cap\mathcal{V}$.*

Proof: All the arguments for the soundness of testEquivSingle are also valid for decideSuppSingle. The test at lines 8–9, as in testEquivSingle, guarantees that the node $n \in v_j$ has a predicate P' into which the predicate P of q maps. The difference is that in decideSuppSingle we are not looking only for predicates P' that are equivalent to a subtree of q; we just check that all predicates of q have an image. Completeness is guaranteed by try all possibilities of finding mappings for the predicates of q. ■

Then, the approach based on view descriptors instead of view definitions remains the same. Regarding support, in order to access the necessary details w.r.t N_v^P, we introduce a new kind a fragment descriptor that records what nodes of q map in an expansion of a tree fragment name. More precisely,

Definition 5.7.2. *For a fragment name f, a set C of /-siblings in q, a set D of //-siblings in q, and a node n from q, a* q-mapping descriptor *denotes a tuple $q\text{-}map(f, C, D, n)$. It says that (i) there exists an expansion v_f which root-maps in the subtree of q rooted at n, (ii) for each $n_i \in C$, there exists a root-mapping m_i of the subtree of q rooted at n_i into v_f, and (iii) for each $n_j \in D$, there exists a mapping m_j of the subtree of q rooted at n_j into v_f.*

Any of the three components C, D or n might be empty.

Note that the space of *q-map* descriptors is exponential in the size of q. They can be computed, in worst-case exponential time, in bottom-up manner straightforwardly.

With these descriptors, we modify **findDescriptors** as follows:

- For the steps (2.b.i) and (3.b.i): if there is a descriptor $q\text{-}map(c_i, C, D, n)$ s.t. $root_P \in C$ and n is a child of $node_q(k_1)$.
- For the step (2.b.ii) and (3.b.ii): if there is a descriptor $q\text{-}map(d_j, C, D, n)$ s.t. $root_P \in D$ and n is a descendant of $node_q(k_1)$.

findDescriptors is sound and complete for computing the descriptors of the QSS expansions. It thus enables a sound and complete algorithm for support (via *testEquivDesc*).

Dealing with compensated views. We use the same $comp(\mathcal{P}, q)$ construction to deal with plans that intersect compensated views.

5.8 QSS with parameters

We consider now an extension to QSS with input parameters for text values (denoted $QSS^{\#}$) and correspondingly, an extension of XP to text conditions. We modify the grammar of XP as follows:

$$pred \quad ::= \quad \epsilon \mid [rpath] \mid [rpath = C] \mid [.//rpath] \mid$$
$$[.//rpath = C] \mid pred\ pred$$

where C terminals stand for text constants. Every node in an XML tree t is now assumed to have a text value *text(t)*, possibly empty. The duality with tree patterns is maintained by associating to every predicate node n in a pattern p a test of equality *test(n)*, that is either the empty word or a constant C. The notions of embedding, mapping and containment can be adapted in straightforward manner to take into account text equality conditions.

The definition of $QSS^{\#}$ can be obtain from Definition 5.2.1 by adding the following: "a leaf element nodes may be additionally labeled with a parameterized equality predicate of the form $= \#i$, where $\#i$ is a *parameter* and i is an integer".

EXAMPLE 5.8.1. *Let us add to $\mathcal{P}$ from Example 5.2.1 the rule*

$$f_1(X) \rightarrow trip[maxprice = \#1]//museum(X)$$

Using this rule, we can generate the view v_4 that retrieves museums on trips for which the maximum price is a parameter #1*:*

$$v_4\text{: } doc(T)//vacation//trip/trip[maxprice=\#1]//museum$$

A user query q_3 that asks for museums with temporary exhibitions on secondary trips that cost at most \$1000

$$q_3\text{: } doc(T)//vacation//trip/trip[maxprice=1000]//museum[temp]$$

is then supported by the QSS, because it can be rewritten as

$$doc(v_4)/v_4/museum[temp](1000)$$

where the parameter #1 *is bound to the value in parenthesis* (1000)*.*

We can show that all the tractability and hardness results presented in the previous sections remain valid when text conditions and parameters are added to the setting. Only a minor adjustment is necessary in order to reuse the exact same PTIME algorithms for expressibility and support (modulo the new XP syntax and the adapted definitions of mapping and containment). Given a query q, the

input $QSS^{\#}$ will be transformed into a QSS $\mathcal{P}'$ by replacing each $= \#i$ parameter occurrence by an explicit text equality condition $= C$, for each constant C appearing in q. Further details are omitted.

5.9 Tractability boundaries

We consider now extensions to the rewrite language and to the query set specifications, asking whether the efficient algorithms of the previous sections can be adapted to deal with them.

Compensated rewriting plans. We consider in this section more complex rewrite plans for support, beyond $XP^{\cap}$, taking the compensation idea one step further. More precisely, we consider the rewrite language $XP^{\cap,c}$ which, after the intersection step, might compensate again for equivalence with the input query. We capture $XP^{\cap,c}$ by adding the following rules to the grammar of XP :

$$\begin{aligned} ipath &::= cpath \mid (cpath) \mid (cpath)/rpath \mid (cpath)//rpath \\ cpath &::= apath \mid apath \cap cpath \end{aligned}$$

Revisiting Definition 3.2.12, a rewriting r in the language $XP^{\cap,c}$ is now of the form $\mathcal{I} = (\bigcap_{i,j} u_{ij})$, $\mathcal{I}/rpath$ or $\mathcal{I}//rpath$, with each u_{ij} being of the form $doc(v_j)/v_j/p_i$ or $doc(v_j)/v_j//p_i$.

EXAMPLE 5.9.1. *Query q_4 below extracts temporary exhibitions from the data about museums visited on the same tour trips as in query q_1:*

q_4: doc(T)//vacation//trip/trip[guide]//tour[schedule//walk]/museum/temp

There is no rewriting of q_4 using only an intersection of views generated by $\mathcal{P}$, since there is no mention of temporary exhibitions in $\mathcal{P}$. However, if we allow the intersection to be compensated, q_4 can be rewritten as the intersection of v_1 and v_2, followed by a one-step navigation:

$$(doc(v_1)/v_1/museum \cap doc(v_2)/v_2/museum)/temp.$$

We prove that support in $XP^{\cap,c}$ becomes NP-hard even for multi-token XP_{es} queries:

Theorem 5.9.1. *For a multi-token XP_{es} query q and a QSS $\mathcal{P}$, deciding if q is supported by $\mathcal{P}$ in $XP^{\cap,c}$ is NP-hard.*

The intuition behind this result is that an $XP^{\cap,c}$ rewriting r for a query q amounts to finding a rewriting r' in the simpler language $XP^{\cap}$ for a prefix of q and then compensating r' with the remainder of q. Even if q were multi-token, r' may correspond to a prefix of q that is in fact single-token, hence the complexity jump.

The proof of Theorem 5.9.1 is similar to the one of Theorem 5.7.1.

We also provide a sound and complete procedure for support on XP_{es} queries in $XP^{\cap,c}$.

To that purpose, we introduce additional notation. A *lossless prefix* p of q is any pattern obtained from a prefix *pref* as q as follows: if q is of the form $pref/t//r$ (where t is not empty and r may be empty), add as a side branch on $\text{OUT}(pref)$ the t pattern. Note that this means that part of the main branch becomes a side branch, hence a predicate. Also note that if q is from the XP_{es} fragment then its lossless prefixes remain in this fragment.

We can prove the following:

Theorem 5.9.2. *An XP_{es} query q is supported by a QSS in $XP^{\cap,c}$ iff some lossless prefix of q is supported by that QSS in $XP^{\cap}$.*

This enables the following EXPTIME decision procedure for support in $XP^{\cap,c}$: test $XP^{\cap}$ support for each lossless prefix, using either the PTIME decision procedure of Section 5.6 (if the prefix is multi-token), or the EXPTIME one of the previous section (if the prefix is single-token).

QSS with forest RHS. We consider now an extension to the query set specifications, which allows *forests* of tree fragments on the RHS, i.e., expansion rules of the form $f \rightarrow tf_1, \ldots, tf_k$.

We call the set specifications in this language QSS^{+}. With this added feature, we show that expressibility and support become NP-hard, even for very restricted tree patterns, without //-edges.

Theorem 5.9.3. *Expressibility is NP-hard for QSS^+, even for XP queries and views without //-edges.*

Proof: We start with expressibility. We detail our proof for boolean tree patterns. The one for patterns of arity 1 is similar.

We use a reduction from the minimum set-cover problem [GJ79]. Let $(\mathcal{U}, \mathcal{S}, k)$ be an instance of this problem, with $\mathcal{U} = \{e_1, \ldots, e_n\}$ denoting the universe, $\mathcal{S} = \{S_1, \ldots S_m\}$ denoting the sets s.t. $S_i \subset \mathcal{U}$ for each S_i. We want to know whether there exists a subset S' of S, of size at most k, that can cover the entire $\mathcal{U}$ (i.e. each element of $\mathcal{U}$ belongs to at least one set of S').

The reduction takes as input the set $\mathcal{U}$ and $\mathcal{S}$ (size $|\mathcal{S}| \times |\mathcal{S}|$) and the value k (size $lg(k)$).

Let p be the biggest exponent s.t. $2^p \leq k$ and let $b_p b_{p-1} \ldots b_0$ be the binary representation of k.

We build the following instance of the expressibility problem. We define the QSS as follows:

- the tree fragments

$$F = \{S, set, f_p^p, f_p^{p-1}, \ldots, f_p^0, f_{p-1}^{p-1}, f_{p-1}^{p-2}, \ldots, f_{p-1}^0, \\ \ldots, f_1^1, f_1^0, f_0^0\}$$

- the alphabet $\Sigma = \{a, e_1, \ldots, e_m\}$
- S is the start fragment
- the set of productions P defined as follows:
  - the start production

$$S \rightarrow a(X)[f_p^p, f_{p-1}^{p-1} \ldots, f_1^1, f_0^0]$$

  - $\forall i = \overline{0, p}$ s.t. $b_i = 1$, we have the productions

$$f_i^k \rightarrow f_i^{k-1}, f_i^{k-1}, \forall k = \overline{0, i}$$

$$f_i^0 \rightarrow set$$

– $\forall i = \overline{0,p}$ s.t. $b_i = 0$, we have the production

$$f_i^i \rightarrow ()$$

– $\forall S_j = \{e_{l_1}, \ldots, e_{l_j}\} \in S$ we have the production

$$set \rightarrow e_{l_1}, \ldots, e_{l_j}$$

– finally we have the production

$$set \rightarrow ()$$

The QSS grammar builds boolean queries having a root node labeled a with children nodes having e-labels. It is easy to see that the generated queries have branches corresponding to a choice of at most k sets from S (outputted by the expansion of the at most k *set* fragment names).

Let now q be the boolean tree pattern $a[e_1][e_2]\ldots[e_n]$.

First, note that all the queries generated by the QSS program contain q. In order for q to be expressed by this program, there must exist a choice of at most k sets from S that covers all the elements, $e_1, \ldots, e_n$. Hence expressibility holds if and only if we can find in S a cover of $\mathcal{U}$ of maximal size k. ■

Theorem 5.9.4. *Support is NP-hard for QSS$^+$, even for XP queries and views without //-edges in predicates.*

Proof: For support, we adapt the previous reduction from the minimum set-cover problem as follows:

The start production is now

$$S(X) \rightarrow b//a[f_p^p][f_{p-1}^{p-1}]\ldots[f_1^1][f_0^0]//c(X),$$

while fragment names expand as before. This program generates views having a main branch $b//a//c$ and having various $[e_i]$ predicates on the a node.

The query q for which we want to test support is

$$b//a[e1]\ldots[e_n]//c.$$

It is easy to see that all the views contain q, hence q is supported if and only if the intersection of *all* the generated views is (i) union-free and (ii) equivalent to q. By using results from Chapter 4, this intersection is union-free if and only if one of the views, call it v, contains all others views. When this is the case, v is equivalent to q if and only if its a node has all the predicates $[e_1], \ldots [e_n]$. But one such view exists if and only if there exists a cover of maximal size k. ■

In the remainder of this section, we also show that expressibility can be solved in exponential time for views encoded by a QSS^+.

5.9.1 Expressibility for QSS^+

We show that the NP lower bound for expressibility is tight for practical purposes, since expressibility can be decided in exponential time. We can prove the following:

Theorem 5.9.5. *Expressibility can be decided in exponential time in the size of the query and of the views.*

We discuss the approach for deciding QSS expressibility (and support) for boolean patterns. Dealing with tree patterns by the same algorithm can be then done in the style of Section 5.3.2.

We now use fragment descriptors defined as follows:

Definition 5.9.1. *For a fragment name f and a set N of nodes from q, a* mapping descriptor *for the pair (f, N) (written map(f, N)) says that there exists an expansion v_f (which can be a tree or a forest) and a mapping m of v_f into q such that the roots of trees in v_f are mapped into the nodes in N.*

Definition 5.9.2. *For a fragment name f, a node $n \in q$ and a set N of same-edge sibling nodes children of n, an* equivalence descriptor *for the pair (f, N) (written equiv(f, N)) says that there exists an expansion v_f (a tree or a forest) satisfying the following:*

- *some $|N|$ trees among those in the forest v_f are equivalent to the subtrees rooted at the nodes of N in q,*

- *if the nodes N are connected by a /-edge to n, the remaining trees in v_f map into q such that the images of their roots are either among the nodes of N or among other /-siblings of the nodes of N,*

- *otherwise (i.e. the nodes N are connected by a //-edge to n), the remaining trees in v_f map into q such that the images of their roots map below n in q.*

Note that in these descriptors we do not bookkeep the number of trees in the expansion v_f. We only keep track of the subtrees of q into which they can map (for mapping descriptors) or of the sibling subtrees in q with which their are equivalent (for equivalence descriptors). Also note that an equivalence descriptor implies a mapping one, for the same fragment name and set of nodes.

The space of distinct descriptors is $O(|\mathcal{G}| \times 2^{|q|})$, hence exponential in the size of the query and polynomial in the size of the program.

5.9.2 Computing descriptors for QSS^+

For a given query q and a QSS $\mathcal{G}$, we can compute all the corresponding descriptors as described below. The computation starts from the productions with no tree fragment nodes and continues inferring descriptors until a fixed point is reached, close in spirit to bottom-up parsing as in the CYK algorithm [HU79] or to bottom-up Datalog evaluation [AHV95]. Each step of this process will run in time worst-case exponential in the size of q and $\mathcal{G}$.

Algorithm findDescriptors:

1. start with an empty set of descriptors D.

2. for each production $f \rightarrow ()$, add to D all the descriptors of $map(f, \{n\})$, for $n \in q$.

3. for each tree production $f \rightarrow tf$, such that tf has only element nodes, compute (by the definitions) and add to D all the possible descriptors $map(f, \{n_q\})$ and $equiv(f, \{n_q\})$.

4. for each tree production $f \to tf$:

   (a) infer new mapping descriptors for f as follows:

   For $f_1, \ldots f_k$ being the tree fragment nodes appearing in tf, for all possible combinations of existing mapping descriptors

   $$c = (map(f_1, N_1), \ldots, map(f_1, N_1)),$$

   let tf_c denote the tree pattern obtained from tf by replacing each tree fragment node f_i by a set of trees that are isomorphic copies of the subtrees of q rooted at the nodes listed in N_i.

   For each mapping ψ of tf_c into q, add to D the descriptor

   $$map(f, \{\psi(\text{ROOT}(tf_c)\}).$$

   Note that by a naive iteration over the space of descriptors this step can be executed in time exponential in the number of tree fragment nodes, k. But a polynomial time approach is possible by dealing with the mapping descriptors in bulk (similar to how mappings algorithms work).

   (b) infer new equivalence descriptors for f as follows:

   Let $c_1, \ldots, c_n$ be the /-children of $\text{ROOT}(tf)$ and let $d_1, \ldots d_m$ be its //-children (either list can be empty). Using their associated descriptors, build a new descriptor $equiv(f, \{n_q\})$ (and implicitly $map(f, \{n_q\})$) for each node $n_q \in q$ such that:

      i. there exist some fragment names $C = \{c_{i_1}, \ldots c_{i_j}\} \subseteq \{c_1, \ldots, c_n\}$ and equivalence descriptors

      $$equiv(c_{i_1}, N_{i_1}), \ldots, equiv(c_{i_j}, N_{i_j}),$$

      such that the set $N_/$ of /-children of n_q satisfies

      $$N_/ = N_{i_1} \cup \cdots \cup N_{i_j},$$

ii. there exist some fragment names $D = \{d_{i_1}, \ldots d_{i_j}\} \subseteq \{d_1, \ldots, d_m\}$, and equivalence descriptors

$$equiv(d_{i_1}, N_{i_1}), \ldots, equiv(d_{i_j}, N_{i_j}),$$

such that the set $N_{//}$ of //-children of n_q satisfies

$$N_{//} = N_{i_1} \cup \cdots \cup N_{i_j},$$

iii. all remaining fragment names $c_i \notin C$ have mapping descriptors of the form $map(c_i, N_q^i)$, for N_q^i being a set of /-children of n_q

iv. all remaining fragment names $d_j \notin D$ have mapping descriptors of the form $map(d_j, N_q^j)$, for N_q^j being a set of descendants of n_q.

The above step can be done by iterating over the descriptors of the cs and ds. Hence this step can be completed in worst-case exponential time in the size of the query and program.

5. for each forest production $f \rightarrow g_1, \ldots g_k$:

(a) for any combination of mapping descriptors

$$map(g_1, N_1), \ldots map(g_k, N_k)$$

add to D the descriptor

$$map(f, N_1 \cup \cdots \cup N_k).$$

(b) for any combination of descriptors

$$c = equiv(g_{i_1}, N_{i_1}), \ldots equiv(g_{i_l}, N_{i_l}),$$
$$map(g_{j_{l+1}}, N_{j_{l+1}}), \ldots, map(g_{i_k}, N_{i_k})$$

for $(i_1, \ldots, i_l, i_{l+1}, \ldots, i_k)$ being any size k permutation, add to D the descriptors

$$equiv(f, N_{i_1} \cup \cdots \cup N_{i_l})$$
$$map(f, N_{i_1} \cup \cdots \cup N_{i_l})$$

when one of the following conditions is verified:

i. all the nodes in $N_{i_1}, \ldots, N_{i_l}, N_{i_{l+1}}, \ldots, N_{i_k}$ are /-siblings in q

ii. all the nodes in $N_{i_1}, \ldots, N_{i_l}$ are //-siblings in q, children of some node n_q, while the nodes in the sets $N_{i_{l+1}}, \ldots, N_{i_k}$ are all somewhere below n_q in q.

This step can executed in time worst-case exponential in the size of the query and program.

6. if any new descriptors have been inferred, go back to step 4.

We can prove the following:

Theorem 5.9.6. *Expressibility holds iff* ***findDescriptors*** *outputs an equivalence descriptor for the start tree fragment, of the form equiv*$(S, \mathrm{ROOT}(q))$. ***findDescriptors*** *runs in exponential time in the size of the QSS and of the query.*

5.10 Related Work

XPath rewriting using only one view [XÖ05, MS05] or a finite, explicitly given set of views [BÖB+04, ABMP07, TYO+08, CDO08] was the object of several studies. To the best of our knowledge, we are the first to address the problem of rewriting XPath queries using a compactly specified set of views. The specifications are written in the Query Set Specification(QSS) language [PDP03], which was also the basis for building a QBE-like XPath interface in a software system for managing biological data [NÖ04]. The QSS language presented in [PDP03] has a different syntax from the one we adopted here and in Appendix 5.B we show how that syntax can be compiled into ours.

Expressibility and support were studied in the past for relational queries and sets of relational views specified by Datalog programs [LRU99, VP00, CDO09]. The work on relational views [CDO09] shares with us the idea of grouping the views in a finite number of equivalence classes w.r.t their behavior in a rewriting algorithm. Similar is also the strategy of computing these classes (represented by what we call *descriptors*) bottom-up from the specification of the sets of views.

However, relational and XPath queries exhibit very different behaviors. For instance, support and expressibility were shown to be inter-reducible in PTIME for relational queries and views [CDO09], and thus share the same complexity (EXPTIME-complete). This is no longer the case for XP queries in the presence of node Ids: expressibility is in PTIME (see Section 5.3), while support is coNP-hard. The PTIME results we obtain make crucial use of the tree shape of XPath queries and require problem-specific restrictions that do not follow from the relational work.

For implementing security policies, a complementary approach to specifying sets of views consists in annotating the DTD of the source with *access annotations* that can be used to allow/disallow access to parts of the input data [FCG04, FGJK07]. The system then infers *one view* over the input document that conforms to the annotations and publishes the DTD of this view. Clients are allowed to ask any queries over the view DTD. This architecture is designed for security scenarios and does not extend to querying sources with limited query capabilities.

APPENDIX TO THE CHAPTER

5.A The function tf-cover

We define here the helper function tf-cover, used in the algorithms for deciding expressibility. tf-cover takes as input a set of nodes N, a set of tree fragment names C and an array L such that for every $n' \in N$, $L(n') \subseteq C$. It returns true if there is a way to pick a distinct tree fragment name from each $L(i)$, for all $i \in N$.

The function is implemented by solving the following max-flow problem with integer values. The flow network has a source s and a sink t. Suppose $C = \{c_1, \ldots c_n\}$ and $L = \{L_1, \ldots L_k\}$. There are edges with capacity 1 from s to n nodes $c_1, c_2, \ldots c_n$. There are also k nodes $L_1, \ldots L_k$ and for each c_j such that $c_j \in L_i$, there is an edge with capacity 1 from c_j to L_i. Finally, there is an edge with capacity 1 from each L_i node to the sink t.

As shown in [CLRS01], if all capacities are integers, the Ford-Fulkerson algorithm will find a max-flow that assigns an integer to every edge.

If the maximum flow returned is less than n (at least one $L(i)$ is not "covered"), tf-cover returns false. Otherwise (the max-flow is n), it returns true. If we want to also keep a view that witnesses expressibility, it is enough to know what edges between a c_j and an L_i have a flow of value 1. This is possible because the Ford-Fulkerson algorithm gives the value of the flow on each edge.

5.B An alternative QSS flavor

In this section, we assume Query Set Specifications given as in [PDP03]. We show that such a QSS can be compiled and normalized in PTIME into the form that can be given as input to our algorithms for expressibility and support. We preprocess an input QSS in two stages. In the first stage, we make the output explicit and in the second one we normalize the specification such that there are no ?, $*$ or $+$ occurrence constraints.

The result of the compilation stage (subsequently used during normalization) is a *QSS with bindings*, which is similar to a QSS except that instead of result node names it uses variables to specify the result node (in the style of Datalog for trees introduced in [AAHM05]). Tree fragment names on the left hand side of a rule may carry a variables bound in the tree fragments from the right hand side. A variable from a tree fragment can either be bound to an element node or to a tree fragment node. If it is an element node, then the variable bindings are given by the matches of that node into a document and that node will be a result node when it appears in an expansion. If it is a tree fragment node f, then its variable bindings come from rules in which f appears on the left hand side. We will also use a syntax of the form

$$f(X) \rightarrow tf_1(Y_1), \ldots tf_k(Y_k)$$

denoting that Y_j is empty or it is a variable bound somewhere in the tree fragment tf_j.

An *expansion* of a QSS with bindings is obtained in a similar way to one for a regular QSS, by replacing non-terminals with the body of the rule in which they appear and keeping the correspondence between variables from the left hand side and from the right hand side.

Theorem 5.B.1. *For any QSS $\mathcal{G}$ there is a QSS with bindings $\mathcal{G}$ and without ?, $*$ or $+$ occurrence constraints such that $\mathcal{G}$ produces the same expansions as $\mathcal{G}'$.*

Proof: We show below the compilation and normalization steps that build $\mathcal{G}'$ from $\mathcal{G}$. ■

5.B.1 Compilation

For the first stage, we start by inferring which tree fragments may contribute to the output and record this using '#' annotations/flags. A tree fragment that contains an element whose name is among the result node names, gets the # and so does the tree fragment name on the left hand side of that production. Then we propagate # flags, until reaching a fix point: if a tree fragment name has #, then all its right-hand-side occurrences in the QSS get # (together with the tree fragment in which they appear); the tree fragment names on the left of those rules also get a #. The number of #'s is at most the number of tree fragments plus the number of rules. And each # can be computed in linear time in the size of the rule in which that tree fragment appears.

Then we replace # flags with variables in the following way.

A rule $f \rightarrow g_1, \ldots, g_n$, in which f has no #, is compiled into $f() \rightarrow g_1(), \ldots, g_n()$.

For a rule $f \rightarrow g_1, \ldots, g_n$ in which f has # set, it is slightly more complicated.

Let $g_{i1}, \ldots, g_{ik}$ be the tree fragments on the right that have #, and contain a tree fragment node that has a #. For each g_{ip} in this set having t tree fragment nodes with #, enumerated in some fixed order, we add the rule

$$f(X_{pi}) \rightarrow g_1(), \ldots, g_{ip}(X_{pi}), \ldots, g_n(), \text{ where } 1 \leq i \leq t$$

where the X_{pi} variable is bound to the i^{th} tree fragment node of g_{ip} having a #.

Let $g_{j1}, \ldots, g_{jm}$ the tree fragments on the right that have # and also have result element nodes.

Then, for every g_{jp} having r result nodes, enumerated in some fixed order, we create r rules

$$f(X_{pi}) \rightarrow g_1(), \ldots, g_{jp}(X_{pi}), \ldots, g_n(), \text{ where } 1 \leq i \leq r$$

in which the variable X_{pi} is bound to the i^{th} result node of g_{jp}.

The number of rules in the grammar obtained through this compilation process may increase by at most the number of nodes in the original grammar. Each new production can be built in linear time in the size of the original production it comes from.

5.B.2 Normalization

At this stage we have a QSS with output specified by variables, such that any tree fragment with ? or $*$ occurrence constraint has no output variable. (Those with + may have output.) We then rewrite this grammar into one without ?, $*$ or +. The transformation rules are given below:

- ? constraint: Any tree fragment node of the form $g()?$ is replaced by a new fragment name $f()$ and the rules below are added:

$$f() \rightarrow g() \tag{5.1}$$

$$f() \rightarrow \tag{5.2}$$

  Any rule containing a $g(X)?$ such that the X is the output of the rule is replaced by a rule in which $g(X)$ has no occurrence constraint.

- $*$ constraint: Any tree fragment node of the form $g()*$ is replaced by a new fragment name *recg*() and the rules below are added:

$$recg() \rightarrow g(), recg() \tag{5.3}$$

$$recg() \rightarrow \tag{5.4}$$

Any rule containing a $g(X)*$ such that the X is the output of the rule is replaced by a rule in which $g(X)$ has no occurrence constraint.

- $+$ constraint:

  Any tree fragment node of the form $g(X)+$ is replaced by a new fragment name $recg(X)$ and the rules below are added:

$$recg(X) \rightarrow g(Y), recg(X) \tag{5.5}$$

$$recg(X) \rightarrow g(X), rech() \tag{5.6}$$

$$recg(X) \rightarrow g(X) \tag{5.7}$$

$$rech() \rightarrow g(Y), rech() \tag{5.8}$$

$$rech() \rightarrow g(Y) \tag{5.9}$$

  where $Y \cap X = \emptyset$ and $rech$ is a new fragment name.

Please note that transformations 5.1-5.9 can be done in linear time in the size of the QSS. In the following, we will consider that our specification is already normalized, i.e. there are no occurrence constraints.

Chapter 5, in part, is a reprint of the material as it will appear in VLDB 2009. Cautis, Bogdan; Deutsch, Alin; Onose, Nicola; Vassalos, Vasilis.

Chapter 6

Contributions to the Data Service Infrastructure

ABSTRACT OF THE CHAPTER

In previous chapters, the main challenge I discussed was how to make services available to the user. Making data sources available as services is a related challenge, as more and more advanced features are required. I extended the standard service infrastructure with new features, several of which were added to the Distributed XQuery (DXQ) framework. DXQ is an XML query and scripting language with support for side effects, distribution, parallelism, which we also used as implementation platform for workflow languages.

This chapter is an overview of my work on building data services. First, I describe how I integrate WSDL and data services and how I optimize service plans with external functions. Then I present my contributions to DXQ.

6.1 Integrating WSDL and Data Services

WSDL represents the standard Web Service interface used in current applications. The first step in creating XQuery-based data services was to define a binding between XQuery and WSDL, described in [OS04] and demoed in [FOS04]. The binding can be used (a) on one hand, to call legacy WSDL services as XQuery

functions and (b) on the other hand, to expose XQuery modules as collections of WSDL services.

I will exemplify part (b) of the binding, as it shows also how Data Services can be implemented as a WSDL extension.

Suppose that the data service from Figure 1.4 is implemented as an XQuery function

```
module namespace app = "http://example.net";
declare function app:getCarByType($type as xs:string)
 as element(Quote)
{ ... };
```

Our xquery2soap tool takes as input this XQuery module and outputs the WSDL interface from Figure 6.1. Some of the details in the WSDL were skipped (replaced by dots), for readability.

The input/output signature of the data service is exposed as input/output *messages* of an *operation* (getCarByType) inside a *PortType* (CarRentalPort). All schema information is grouped inside the types element. Protocol specific details, and the URL at which the service can be called are published inside the binding and service WSDL elements.

To consume the CarRental services, users would either send a SOAP call using any WSDL/SOAP bindings written for general purpose programming languages (C#, Perl, Java etc.) or they would call it from withing XQuery using our *import service* extension:

```
import service namespace app = "http://example.net" name "CarRental";
app:getCarByType("Sedan")
```

This extension, described in [OS04], is just a minimal add-on to our XQuery implementation (Galax [Gal]), as it is internally compiled into a regular module import.

The WSDL-XQuery binding provides thus a way to publish the input/output signature of a data service in a WSDL format. Publishing a data service, as we have seen in Chapter 1, also requires publishing the query that defines its public behavior. This can be easily done by embedding the XQuery code inside the WSDL

```
<definitions targetNamespace="http://example.net"
xmlns:tns="http://example.net" ...>

 <types>
  <xs:schema targetNamespace="http://example.net">
    ...
    <xs:element name="Quote">
      <xs:complexType>...</xs:complexType>
    </xs:element>
  </xs:schema>
 </types>

 <message name="getCarByType">
   <part name="type" type="xs:string"/>
 </message>

 <message name="getCarByTypeResponse">
  <part name="result" element="tns:contact"/>
 </message>

 <portType name="CarRentalPort">
  <operation name="getCarByType">
   <input message="tns:getContact"/>
   <output message="tns:getContactResponse"/>
  </operation>
 </portType>

 <binding name="CarRentalSOAP"
  type="tns:UserProfilePort">
   <soap:binding style="rpc"
    transport="schemas.xmlsoap.org/..."/>
   <operation name="getCarByType">
    ...
   </operation>
 </binding>

 <service name="CarRental">
   <port name="CarRentalPort"
        binding="tns:UserProfileSOAP">
    <soap:address
     location="http://example.net/services/app.xqs"/>
   </port>
 </service>

</definitions>
```

Figure 6.1: WSDL interface for the car rental data service

document. Moreover, it does not go against the current WSDL specifications, as WSDL allows for arbitrary extensions.

6.2 Query Plans with External Function Calls

Data service developers need to be able to call functions implemented in other programming languages, either because they need features that are not supported by XQuery or because they want to re-use function libraries written in a general purpose programming language.

A well-known framework for developing data services is the BEA AquaLogic Data Services Platform [BCL+06] (ALDSP). An ALDSP application consists of a hierarchy of functional XQuery views. The base of the hierarchy is made up of physical services that are wrappers over the data sources. In order to leverage the power of underlying database systems, one has to be able to take advantage of their query capabilities. But many applications perform transformations, typically changes of data format, using functions that are not supported by the sources. This leads to producing non transparent query plans that cannot be pushed to the database engine and require that more work be done on the mediator, which usually has less resources for processing large amounts of data.

My contribution in this domain was to add the capacity of rewriting expressions that contain externally defined functions by registering corresponding inverse functions and transformations that define where the inverses can be used [OBC07, BCL+06]. Thus, the query optimizer is able to infer equivalent query plans and to choose the one which is better suited for the capabilities of each source.

6.3 Distributed XQuery(DXQ)

The goal of the DXQ project [FJM+07b, FJM+07a] is to support development of reliable, extensible, and efficient distributed resource-management protocols. Our strategy to meet these requirements is to provide a high-level, distributed, and optimizable query language for implementing distributed resource-management protocols. By using a high-level language, a protocol's semantics is transparent, not hidden in the implementation, which supports the reliability requirement. In addition, the implementation is optimizable by general query opti-

mization techniques. Automating optimization supports the efficiency requirement and permits implementors to focus more on functionality and less on performance.

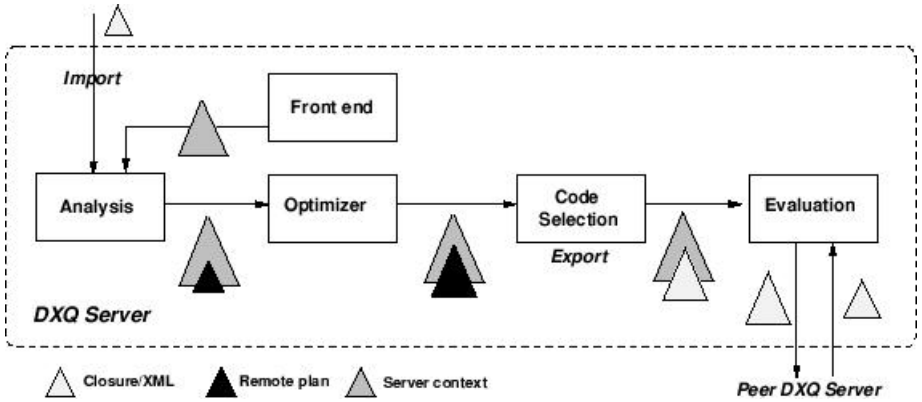


Figure 6.2: Galax XQuery architecture

DXQ is implemented in the Galax XQuery architecture (Figure 6.2), which compiles XQuery programs into an algebraic representation (a query plan) that is exchanged by DXQ servers. This plan is enclosed into a remote closure which allows to ship an arbitrary expression and encapsulates whatever context is necessary to evaluate the expression remotely. Galax's hybrid algebra includes XML "tree" operators and classic tuple operators. The latter permit efficient implementation (the Optimizer stage) of common query idioms like join an group-by, which are expressed by nested FLWOR expressions in XQuery. The optimized plan is evaluated based on the code selection performed on the server.

The DXQ extension of Galax relies on a simple client-server protocol based on HTTP that permits the communication between DXQ peers. I will not go into the details of the network infrastructure, explained in [FJM+07a], and instead I will present two related projects. The first one consists in the in-depth study and implementation of XML updates, which we used for supporting DXQ applications. The second one is a compiler and a runtime architecture that allows running workflows with the DXQ engine. The ability to run workflows proved the versatility of the DXQ platform and its potential for creating rich applications.

6.3.1 XML Updates

In the study of expressibility and support in this dissertation, I considered service calls without side effects. However, data services in general, and DXQ applications in particular, do not perform only queries, but also data updates. To provide a functional data service platform, my goal was then to enable query plan optimizations over data services executing both queries and data updates.

The challenge is that most traditional compile rewrites rely on query plans being free of side effects. Otherwise, rewritings may lead to incorrect results. For instance, the input of an algebraic operator might not be the same or the number of updates performed may change.

Rewriting algorithms that consider updates are also useful if we try to push the study of expressibility and support further. When rewriting queries using services that perform updates, changing the order of the service calls may lead to different results for the reasons mentioned above.

The approach I chose was to test the soundness of a rewriting based on static analysis. Rewrite rules that are provably sound can then be used to optimize query plans that contain updates.

Some examples of basic static checks are:

- the evaluation of two subplans can change (*commutativity*),
- applying an operator once has the same effect as applying it twice (*idempotency*),
- absence of side effects (*purity*).

Every optimization rule has a set of such conditions that guarantee its correctness.

We gave the first formalization of an algebra with updates [GORS07] to lay the theoretical grounds for research on query plans with side effects. Then, in our SIGMOD paper [GORS08], we showed how we can recover a large class of database optimizations in the presence of updates. These include most classical logical optimizations, such as join detection or unnesting, and pipelining.

6.3.2 Implementing Workflows as Data Services

Workflow languages are the norm when it comes to representing and implementing business processes. With the emergence of Web-enabled workflow languages, such as BPEL [OAS07], there is an increasing need to support XML processing along with those languages. I extended the REST-based [Fie00] workflow language Bite (that implements the basic functionalities of BPEL) with XQuery processing capabilities. The resulting language can be implemented on top of a stand-alone XQuery processor by compiling its core constructs into DXQ, a distributed extension of XQuery. From an XQuery perspective, this approach demonstrates the expressiveness of the DXQ framework. From a workflow perspective, it opens interesting opportunities for light-weight implementations of Web workflows, cross-activity optimization, and experimentation with distributed workflows.

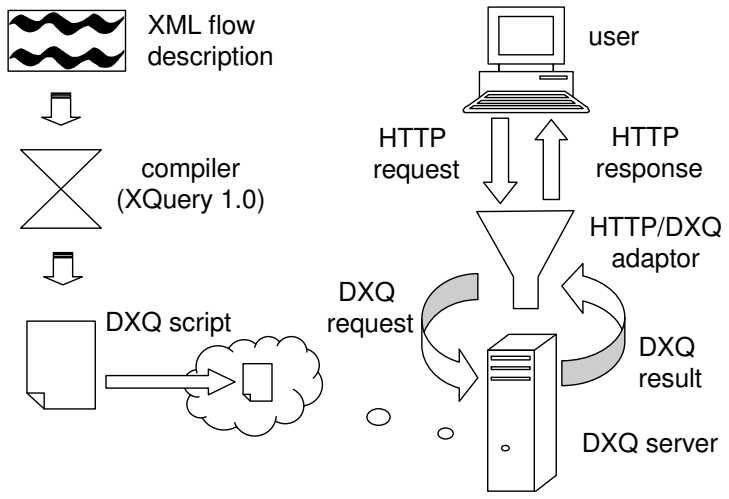


Figure 6.3: Bite+XQuery: Compilation and Runtime Architecture

Figure 6.3 gives an overview of how we compile and run workflows written in the XQuery extension of Bite. The compiler implementation consists in 915 lines of standard XQuery 1.0, that I tested with two distinct open source XQuery processors: Galax [Gal] and Saxon [Sax]. The compiler takes as input a description of the flow, in XML format, and produces a DXQ module that implements it. The next step is to generate a special *dispatch* function that accepts the requests sent

to a flow and forwards them to the appropriate workflow instance (if the requests refer to an existing instance) or creates a new instance.

The target platform of our compiler is the DXQ runtime. After compiling a workflow specification, the resulting DXQ module and interface are run as any DXQ server. In order to provide an external interface compatible to the protocol used by Bite (pure REST), I added an adaptor which acts as a proxy between the HTTP clients and the DXQ runtime. The adaptor is also used to translate non-XML input, such as the one coming from an HTML form, into an XML encoding. The reader is referred to our XIME-P paper [OKRS08] for a more detailed description of this project.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Data services are able to simplify the development and maintenance for current data-centric Web applications and can enable new features required for publishing and discovering services that interact with data sources.

The central theme of my dissertation is the study of two problems faced by Web developers that need to use data services expressed as queries over data sources. The first one is expressibility (deciding whether a user request is equivalent to a published service) and the second one is support (deciding whether a user request is equivalent to a query over the published services). I study these problems both in the case in which the services are listed individually and when the set of services published by the source is compactly encoded by a specification such as QSS or Datalog.

For listed sets of services defined by queries, expressibility is the same as query equivalence, which has already been extensively studied in previous literature. Support for XML services on the other hand, reduces to the problem of rewriting XML queries using views, for which previous solutions were either limited to very simple classes of queries or were based on heuristics without a formal study of their applicability.

I present a decision procedure for support that covers a large fragment of XQuery, including FLWR expressions, equality conditions, existential quanti-

fiers. This is done by adopting a pattern based formalism called the Nested XML Tableaux (NEXT). Pattern based representations were also the key for success in previous research on equivalence and rewriting queries using views: conjunctive queries and tableaux were used for relational queries, tree patterns for XPath queries.

When going to the full XQuery language, that I represent as NEXT+ patterns, both expressibility and support become undecidable. The good news is that the rewriting algorithm I propose for NEXT queries extends naturally to a sound test for NEXT+. Its abilities to find rewritings can be improved by adding rules that define properties of operators or functions used in the queries.

One operator that deserves to be studied in its right is intersection, as it is traditionally a basic feature of database query languages. However, while for relational queries, using intersection in the rewritten queries does not change the complexity of support, XML queries exhibit a different behavior. Rewriting tree pattern (XPath) queries using tree pattern views is PTIME in the absence of intersection, and I show that it is coNP-complete when the rewrite plan can intersect views. I also show that the related problem, of deciding if an intersection of XPath queries can be equivalently rewritten as one XPath, without intersection or union, is hard. I give syntactic restrictions, applicable to large classes of tree patterns, under which these problems can be solved in PTIME.

For the case of compactly encoded sets of views, previous work [LRU99, VP00] studied expressibility and support for relational queries and views specified as expansions of a Datalog program. Continuing this line of work, I go one step further and look at expressibility and support for infinite sets of relational views, under constraints on the data source. I identify conditions under which the two problems are decidable and propose algorithms that are sound in general and complete under the decidability restrictions. The algorithm for support, if used in the absence of constraints, yields an exponential factor improvement over previous algorithms.

The next step is to consider compactly encoded XML queries, as most Web applications manipulate XML data. I study the problems of expressibility and

support of an XPath query by XPath views generated as expansions of a Query Set Specification. Since I focus on efficiency, I consider only PTIME algorithms, ensuring that they are sound in general and identifying the most permissive restrictions under which they become complete. I find that for XPaths corresponding to the fragment having child and descendant navigation and no wildcard, expressibility can be solved in PTIME. For support, the complexity analysis is more refined, as it depends on the rewriting language. Therefore I consider both the case in which the ids of the nodes are lost when constructing the query result and the case in which the ids are persistent. For the former case, I give a PTIME decision procedure. For the latter, I propose a sound PTIME algorithm that also becomes complete under restrictions.

7.2 Future Work

In the future, the applicability of the techniques I developed can be gradually enlarged to broader classes of data services. This includes extending the work on rewriting XPath queries using an intersection of XPath views to the entire XQuery language. This can be done by designing a sound test for support that uses the algorithm for XPath to rewrite the navigational part of the queries. Then the algorithms for expressibility and support by sets of QSSL encoded views could also be extended from XPath to XQuery. Such a succession follows the approach I have taken until now of grounding the study of compactly encoded views on results and intuition coming from the case of explicitly listed views.

Besides data processing, data services typically also perform data updates. How to publish sets of data services (either listed or compactly encoded) defined by a mix of queries and updates is then a natural question. To study expressibility and support in the presence of updates, my research on optimizing query plans that contain updates [GORS08] is a starting point when looking for intuition and useful techniques. For instance, the statically checkable conditions from [GORS08], when applied to plans over data service calls, may enable rewritings that change the order of updates or their position in the plan.

As applications become more sophisticated, data services need to become richer in features, and eventually able to express workflows. An example of such workflows are the XQuery-enabled business processes ran by the XQuery-based prototype I implemented over an XML query and scripting framework [OKRS08]. An important challenge, left to future work, is then to extend tests for expressibility and support (and the accompanying tools for producing the actual rewritten plans) to services implemented as workflows that perform data manipulations. Since language features present in workflows and in XQuery easily lead to Turing completeness, any algorithm would then necessarily be incomplete, i.e., it may return false negatives. But this was already the case for rewriting just XQuery queries using XQuery views.

From a software engineering perspective, such an algorithm for rewriting services defined as workflows can become the centerpiece of a tool for automatic composition of data-centric workflows that takes into account the data manipulations they perform. Extending the algorithm and the tool to deal also with scientific workflows is an additional challenge, motivated by the development of Web-distributed scientific data sets and applications.

Bibliography

[AAHM05] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS*, pages 358–367, 2005.

[ABMP07] Andrei Arion, Véronique Benzaken, Ioana Manolescu, and Yannis Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, pages 87–98, 2007.

[ACGP06] Foto N. Afrati, Rada Chirkova, Manolis Gergatsoulis, and Vassia Pavlaki. Finding equivalent rewritings in the presence of arithmetic comparisons. In *EDBT*, pages 942–960, 2006.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[ALM02] Foto N. Afrati, Chen Li, and Prasenjit Mitra. Answering queries using views with arithmetic comparisons. In *PODS*, 2002.

[AYCLS02] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4), 2002.

[BBC+07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML path language (XPath) 2.0, 2007.

[BCF+] Scott Boag, Don Chamberlain, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language, W3C candidate recommendation 8 june 2006. http://www.w3.org/TR/2006/CR-xquery-20060608/.

[BCL+06] Vinayak R. Borkar, Michael J. Carey, Dmitry Lychagin, Till Westmann, Daniel Engovatov, and Nicola Onose. Query processing in the AquaLogic Data Services Platform. In *VLDB*, pages 1037–1048, 2006.

[BDF+01] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang chiew Tan. Reasoning about keys for xml. In *Information Systems*, page 2003, 2001.

[BEA] BEA Systems. Data services developer's guide. best practices and advanced topics. http://edocs.bea.com/aldsp/docs21/datasrvc/bestpractices.html.

[BFG05] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, pages 25–36, 2005.

[BFK05] Michael Benedikt, Wenfei Fan, and Gabriel Kuper. Structural properties of XPath fragments. *Theor. Comput. Sci.*, 336(1):3–31, 2005.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.

[BÖB+04] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[BPSM+06] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (fourth edition), 2006.

[CAM07] Bogdan Cautis, Serge Abiteboul, and Tova Milo. Reasoning about XML update constraints. In *PODS*, pages 195–204, 2007.

[Car06] Michael Carey. Data delivery in a service-oriented world: the BEA AquaLogic Data Services Platform. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 695–705, New York, NY, USA, 2006. ACM.

[CCMW] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[CD99] James Clark and Steve DeRose. XML path language (XPath), 1999.

[CDO08] Bogdan Cautis, Alin Deutsch, and Nicola Onose. XPath rewriting using multiple views: Achieving completeness and efficiency. In *WebDB*, 2008.

[CDO09] Bogdan Cautis, Alin Deutsch, and Nicola Onose. Querying data sources that export infinite sets of views. In *ICDT*, pages 84–97, 2009.

[CFF+] Don Chamberlain, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. W3C Working Draft 8 June 2006. http://www.w3.org/TR/2006/WD-xquery-use-cases-20060608/.

[CFF+06] Michael J. Carey, Mary Fernndez, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: An XML application development language. In *XIME-P*, 2006.

[CFM+08] Don Chamberlain, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. XQuery Update Facility 1.0, 2008.

[CGKV88] Stavros Cosmadakis, Haim Gaifman, Paris Kanellakis, and Moshe Vardi. Decidable optimization problems for database logic programs. In *STOC*, pages 477–490, New York, NY, USA, 1988. ACM Press.

[CKS+00] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. Xperanto: Middleware for publishing object-relational data as xml documents. In *VLDB*, pages 646–648, 2000.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[CNS06] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst. (TODS)*, 31(2):672–715, 2006.

[CR97] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 1997.

[CR02] Li Chen and Elke A. Rundensteiner. XCache: XQuery-based caching system. In *WebDB*, pages 31–36, 2002.

[CS99] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.

[CV93] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of *eal* conjunctive queries. In *PODS*, pages 59–70, 1993.

[dBGV05] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. Well-definedness and semantic type-checking in the nested relational calculus and xquery extended abstract. In *ICDT*, pages 99–113, 2005.

[DFF+] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics, W3C candidate recommendation 8 june 2006. http://www.w3.org/TR/2006/CR-xquery-semantics-20060608/.

[DFK+04] Yanlei Diao, Daniela Florescu, Donald Kossmann, Michael J. Carey, and Michael J. Franklin. Implementing memoization in a streaming XQuery processor. In *XSym*, pages 35–50, 2004.

[DHT04a] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested XML queries. In *VLDB*, pages 132–143, 2004.

[DHT04b] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested xml queries. In *VLDB*, pages 132–143, 2004.

[DLN05] Alin Deutsch, Bertram Ludaescher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. In *ICDT*, 2005.

[DLN07] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.

[DPT06] Alin Deutsch, Lucian Popa, and Val Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[DPX04] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The NEXT logical framework for xquery. In *VLDB*, pages 168–179, 2004.

[DT03a] Alin Deutsch and Val Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, pages 201–212, 2003.

[DT03b] Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In *ICDT*, pages 225–241, 2003.

[FCG04] Wenfei Fan, Chee Yong Chan, and Minos N. Garofalakis. Secure XML querying with security views. In *SIGMOD Conference*, pages 587–598, 2004.

[FFG01] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. In Jan Van den Bussche and Victor Vianu, editors, *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, volume 1973 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2001.

[FFM03] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the minimization of XPath queries. In *VLDB*, pages 153–164, 2003.

[FGJK07] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Rewriting regular xpath queries on xml views. In *ICDE*, pages 666–675, 2007.

[Fie00] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, UC Irvine, 2000.

[FJM+07a] Mary F. Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jérôme Siméon. Dxq: a distributed xquery scripting language. In *XIME-P*, 2007.

[FJM+07b] Mary F. Fernández, Trevor Jim, Kristi Morton, Nicola Onose, and Jérôme Siméon. Highly distributed xquery with dxq. In *SIGMOD Conference*, pages 1159–1161, 2007.

[FKMP03] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[FLMS99] Daniela Florescu, Alon Y. Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD*, pages 311–322, 1999.

[FOS04] Mary F. Fernández, Nicola Onose, and Jérôme Siméon. Yoo-hoo! building a presence service with XQuery and WSDL. In *SIGMOD Conference*, pages 911–912, 2004.

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer second edition, 2004.

[Gal] Galax: An implementation of XQuery. http://www.galaxquery.org.

[GBG06] Sven Groppe, Stefan Böttcher, and Jinghua Groppe. XPath query simplification with regard to the elimination of intersect and except operators. In *ICDE Workshops*, page 86, 2006.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GKP03] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190, 2003.

[GMSV93] Haim Gaifman, Harry G. Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM (JACM)*, 40(3):683—713, 1993.

[GORS07] Giorgio Ghelli, Nicola Onose, Kristoffer Høgsbro Rose, and Jérôme Siméon. A better semantics for XQuery with side-effects. In *DBPL*, pages 81–96, 2007.

[GORS08] Giorgio Ghelli, Nicola Onose, Kristoffer Høgsbro Rose, and Jérôme Siméon. XML query optimization in the presence of side effects. In *SIGMOD Conference*, pages 339–352, 2008.

[GRS06] Giorgio Ghelli, Christopher Re, and Jérôme Siméon. XQuery!: An XML query language with side effects. In *EDBT Workshops*, pages 178–191, 2006.

[GWY07] Jun Gao, Tengjiao Wang, and Dongqing Yang. MQTree based query rewriting over multiple XML views. In *DEXA*, pages 562–571, 2007.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[Hid03] Jan Hidders. Satisfiability of XPath expressions. In *DBPL*, pages 21–36, 2003.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[KM90] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. VLDB*, Brisbane, Australia, 1990.

[LC00] Chen Li and Edward Y. Chang. Query planning with limited source capabilities. In *ICDE*, pages 401–412, 2000.

[LMSS95a] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of PODS*, 1995.

[LMSS95b] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[LRU96] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external processors. In *PODS*, pages 227–237, 1996.

[LRU99] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external query processors. *J. Comput. Syst. Sci.*, 58(1):69–82, 1999.

[LS97] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects. In *PODS*, pages 20–31, 1997.

[LWZ06] Laks V. S. Lakshmanan, Hui Wang, and Zheng Zhao. Answering tree pattern queries using views. In *VLDB*, pages 571–582, 2006.

[MFK01] I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries on heterogeneous data sources. In *VLDB*, 2001.

[MMS79] Albert Maier, Alberto Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. In *PODS*, 1979.

[Mot89] Amihai Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*, pages 339–347. IEEE Computer Society, 1989.

[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*, pages 469–480, 2005.

[NL04a] A. Nash and B. Ludäscher. Processing unions of conjunctive queries with negation under limited access patterns. In *EDBT*, 2004.

[NL04b] Alan Nash and Bertram Ludäscher. Processing first-order queries under limited access patterns. In *PODS*, 2004.

[NÖ04] Scott Newman and Z. Meral Özsoyoglu. A tree-structured query interface for querying semi-structured data. In *SSDBM*, pages 127–130, 2004.

[NS03] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, pages 312–326, 2003.

[NS06] Frank Neven and Thomas Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*, 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[OBC07] Nicola Onose, Vinayak R. Borkar, and Michael J. Carey. Inverse functions in the aqualogic data services platform. In *VLDB*, pages 1231–1242, 2007.

[ODPC06] Nicola Onose, Alin Deutsch, Yannis Papakonstantinou, and Emiran Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD Conference*, pages 443–454, 2006.

[OKRS08] Nicola Onose, Rania Khalaf, Kristoffer Høgsbro Rose, and Jérôme Siméon. A restful workflow implementation on top of distributed XQuery. In *XIME-P*, 2008.

[OS04] Nicola Onose and Jerome Simeon. XQuery at your web service. In *Proceedings of the 13th international World Wide Web conference (WWW'04)*, pages 603–611, New York, NY, USA, 2004.

[OWL] OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

[PDP03] Michalis Petropoulos, Alin Deutsch, and Yannis Papakonstantinou. The Query Set Specification Language (QSSL). In *WebDB*, pages 99–104, 2003.

[PGGMU95] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, and Jeffrey D. Ullman. A query translation scheme for rapid implementation of wrappers. In *DOOD*, pages 161–186, 1995.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *SIGMOD Conference*, pages 455–466, 1999.

[Ram02] Prakash Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD Conference*, pages 299–309, 2002.

[RMSR04] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, page 14, 2006.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112. ACM Press, 1995.

[RSUV89] Raghu Ramakrishnan, Yehoshua Sagiv, Jeffrey D. Ullman, and Moshe Y. Vardi. Proof-tree transformation theorems and their applications. In *PODS*, pages 172–181, 1989.

[Sax] Saxon: The XSLT and XQuery processor. http://saxon.sourceforge.net.

[SV05] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In *PODS*, pages 49–60, 2005.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[tCL07] Balder ten Cate and Carsten Lutz. The complexity of query containment in expressive fragments of XPath 2.0. In *PODS*, pages 73–82, 2007.

[TYO+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.

[TZ05] Jian Tang and Shuigeng Zhou. A theoretic framework for answering XPath queries using views. In *XSym*, pages 18–33, 2005.

[UH79] J. .D Ullman and J. E. Hopcroft. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[Val87] P. Valduriez. Join indices. *ACM Trans. Database Systems*, 12(2):218–452, June 1987.

[Van05] Stijn Vansummeren. Deciding well-definedness of XQuery fragments. In *PODS*, pages 37–48, 2005.

[VP97] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *VLDB*, pages 256–265, 1997.

[VP00] Vasilis Vassalos and Yannis Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. *J. Log. Program.*, 43(1):75–122, 2000.

[Woo03] Peter T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, pages 297–311, 2003.

[XÖ05] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, pages 121–132, 2005.

[Xu05] Yu Xu. *The NEXT+ Framework for Logical XQuery Optimization.* PhD in Computer Science, University of California, San Diego, 2005.

[YLH03] Liang Huai Yang, Mong Li Lee, and Wynne Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, pages 69–80, 2003.

[YP04] Cong Yu and Lucian Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD Conference*, pages 371–382, 2004.