Institut für Informatik

Ludwig Maximilians Universität München

# IMPLEMENTATION OF WEB QUERY LANGUAGES RECONSIDERED

## BEYOND TREE AND SINGLE-LANGUAGE ALGEBRAS AT (ALMOST) NO COST

TIM FURCHE


MUNICH 2008

Institut für Informatik

Ludwig Maximilians Universität München

LMU

# IMPLEMENTATION OF WEB QUERY LANGUAGES RECONSIDERED

## BEYOND TREE AND SINGLE-LANGUAGE ALGEBRAS AT (ALMOST) NO COST

TIM FURCHE

DISSERTATION

an der Fakultät für Mathematik, Informatik und Statistik

der Ludwig-Maximilians Universität München

München

vorgelegt von

TIM FURCHE

aus Tübingen

München, den 10. April 2008

| | |
|---|---|
| ERSTGUTACHTER: | FRANÇOIS BRY |
| ZWEITGUTACHTER: | GEORG GOTTLOB |
| Tag der MÜNDLICHEN PRÜFUNG: | 8. MAI 2008 |

Tim Furche: *Implementation of Web Query Languages Reconsidered—Beyond Tree and Single-Language Algebras at (almost) No Cost,* PH.D. thesis, © 8. Mai 2008

# ABSTRACT

Visions of the next generation Web such as the "Semantic Web" or the "Web 2.0" have triggered the emergence of a multitude of data formats. These formats have different characteristics as far as the shape of data is concerned (for example tree- vs. graph-shaped). They are accompanied by a puzzlingly large number of query languages each limited to one data format. Thus, a key feature of the Web, namely to make it possible to access anything published by anyone, is compromised.

This thesis is devoted to versatile query languages capable of accessing data in a variety of Web formats. The issue is addressed from three angles: language design, common, yet uniform semantics, and common, yet uniform evaluation.

First, we consider the query language Xcerpt as an example of the advocated class of *versatile Web query languages*. Using this concrete exemplar allows us to clarify and discuss the vision of versatility in detail.

Second, a number of query languages, XPath, XQuery, SPARQL, and Xcerpt, are translated into a *common intermediary language*, CIQLOG. This language has a purely logical semantics, which makes it easily amenable to optimizations. As a side effect, this provides the, to the best of our knowledge, first logical semantics for XQuery and SPARQL. It is a very useful tool for understanding the commonalities and differences of the considered languages.

Third, the intermediate logical language is translated into a *query algebra*, CIQCAG. The core feature of CIQCAG is that it scales from tree- to graph-shaped data and queries without efficiency losses when tree-data and -queries are considered: it is shown that, in these cases, optimal complexities are achieved. CIQCAG is also shown to evaluate each of the aforementioned query languages with a complexity at least as good as the best known evaluation methods so far. For example, navigational XPath is evaluated with space complexity $\mathcal{O}(q \cdot d)$ and time complexity $\mathcal{O}(q \cdot n)$ where $q$ is the query size, $n$ the data size, and $d$ the depth of the (tree-shaped) data.

CIQCAG is further shown to provide linear time and space evaluation of tree-shaped queries for a larger class of graph-shaped data than any method previously proposed. This larger class of graph-shaped data, called *continuous-image graphs*, short CIGS, is introduced for the first time in this thesis. A (directed) graph is a CIG if its nodes can be totally ordered in such a manner that, for this order, the children of any node form a continuous

interval.

CIQCAG achieves these properties by employing a novel data structure, called *sequence map*, that allows an efficient evaluation of tree-shaped queries, or of tree-shaped cores of graph-shaped queries on any graph-shaped data. While being ideally suited to trees and CIGs, the data structure gracefully degrades to unrestricted graphs. It yields a remarkably efficient evaluation on graph-shaped data that only a few edges prevent from being trees or CIGs.

## ZUSAMMENFASSUNG

Zukunftsvisionen über das Web der Zukunft, angepriesen unter Begriffen wie dem „Semantische Web“ oder dem „Web 2.0“, haben in den letzten Jahren die Entwicklung einer Vielzahl von neuen Datenformaten ausgelöst. Nicht nur unterscheiden sich diese Formate im Hinblick auf die erlaubte Struktur (beispielsweise Baum- vs. Graph-Daten), sie haben auch eine Myriade von Anfragesprachen mit sich gebracht, zumeist auf jeweils ein Format eingeschränkt. Damit droht eine zentrale Eigenschaft des Webs, die Fähigkeit auf jedwede Information zugreifen, wer immer sie veröffentlicht hat, verloren zu gehen.

Um dieses Problem anzugehen, ist diese Arbeit „versatilen“, also vielseitig einsetzbaren, Anfragesprachen für das Web gewidmet. Vielseitigkeit wird aus drei Perspektiven betrachtet: Sprachentwurf, Semantik, und Auswertung. Diese drei Perspektiven dienen als Leitfaden für die gesamte Arbeit:

Zu Beginn wird die Anfragesprache Xcerpt als ein Beispiel für die Klasse der „versatilen“ Anfragesprachen vorgestellt und eine Reihe von konkreten Einsatzszenarien für solche Anfragesprachen diskutiert.

Der zweite Teil der Arbeit zeigt auf, wie eine Reihe von Anfragesprachen, genauer XPath, XQuery, SPARQL und Xcerpt, in eine einheitliche Zwischensprache, genannt CIQLog, übersetzt werden kann. CIQLog hat eine rein logische Semantik, die eine ideale Basis für Optimierungen und algebraische Umschreibungen bietet wie im dritten Teil gezeigt. Ein Seiteneffekt der Übersetzung ist die, unseres Wissens nach, erste rein logische Semantik für XQuery und SPARQL. CIQLog entpuppt sich auch als hervorragende Basis für die Untersuchung von Ähnlichkeiten wie Besonderheiten der betrachteten Sprachen.

Der zentrale Beitrag der Arbeit wird im dritten Teil vorgestellt: die CIQCAG Anfrage-Algebra, die zur Implementierung von CIQLog (und damit aller oben genannten Sprachen) verwendet wird. Im Zentrum von CIQCAG

steht die Fähigkeit mit unterschiedlich strukturierten Daten und Anfragen gleichermaßen effizient umgehen zu können: Insbesondere, zeigt die Arbeit, dass die Auswertung mit CIQCAG sowohl für Baum- als auch für Graph-Daten optimale Komplexität besitzt. CIQCAG ist dadurch imstande jede der oben genannten Anfragesprachen wenigstens so gut wie die besten bisher bekannten Ansätze auswerten zu können. Navigationelles XPath, beispielsweise, kann durch CIQCAG mit $\mathcal{O}(q \cdot d)$ Speicher- und $\mathcal{O}(q \cdot n)$ Zeitkomplexität ausgewertet werden (dabei ist $q$ die Anfrage-, $n$ die Datengröße und $d$ die Tiefe der (Baum-) Daten).

Schließlich erlaubt CIQCAG die Auswertung von Baum-Anfragen mit linearer Zeit und linearem Speicher auf einer größeren Klasse von Graph-Daten als alle bisherigen Ansätze. Diese Klasse von Graph-Daten, genannt „*continuous image graphs*“, kurz CIGS, wird in dieser Arbeit erstmalig vorgestellt. Ein Graph fällt in diese Klasse, sobald seine Knoten so geordnet werden können, dass, über diese Ordnung, die Kinder jedes Knotens ein kontinuierliches Intervall bilden.

Diese Eigenschaften erreicht CIQCAG durch den Einsatz einer neuartigen Datenstruktur, genannt *sequence map*, die die effiziente Auswertung von Baum-Anfragen (oder -Teilanfragen) auf beliebigen Graph-Daten erlaubt. Die Datenstruktur ist ideal für Bäum- und CIG-Daten geeignet, kann aber auch für allgemeine Graph-Daten eingesetzt werden. Dabei erhält man eine bemerkenswert effiziente Auswertung, wenn die Graphen nahezu die Form von Bäumen oder CIGS haben mit nur wenigen abweichenden Kanten.

# ACKNOWLEDGMENTS

This thesis has been a source of great joy for me, despite the fact that it has dominated most of my professional (and, all too often, other) life for a good four years. Just as this thesis, none of that joy would have been possible without a great many persons in my life. I am deeply ingratiated to all those persons, but can only highlight a select few here.

Undoubtedly, the person that has been the greatest source of ideas, knowledge, support, and encouragement has been François Bry, my thesis advisor. I am deeply thankful to him for his patience and open-mindedness without loosing sight of what needs to be done, hundreds of discussions on topics of the thesis, life as a researcher, and about anything else, and, last but not least, his leadership in the REWERSE project that has not only funded my position but also allowed me to make contacts and friends with researchers from all over Europe.

The two persons that have taught me most about how to do research and have fun doing it, are certainly Dan Olteanu, with whom I spend incredible years working on XPath and SPEX, and Tim Geisler, who introduced me to the world of XML and the joy of teaching. With the constant support of Norbert Eisinger, I have been able to preserve that joy even in the face of contradictory or even nonsensical regulations, barely working administrations, and crumbling buildings.

They are just three examples of the incredible luck I have had regarding my colleagues in the Munich group of François. The work with Sacha Berger, Clemens Ley, Benedikt Linse, Andreas Schroeder, and Antonius Weinzierl has not only been immensely fruitful for this thesis, I have also enjoyed working, talking, and just being around you a whole lot. The same applies for Michael Eckert, Andreas Häusler, Alex Kohn, Michael Kraus, Bernhard Lorenz, Hans-Jürgen Ohlbach, Paula-Lavinia Pătrânjan, Martin Sachenbacher, Sebastian Schaffert, Stephanie Spranger, Edgar Stoffel, Felix Weigel, and Christoph Wieser.

Undeservedly, the REWERSE project has allowed me to be part of another great "family" of researchers including Uwe Aßmann, who taught me much about life as a researcher (and how to get lost in Lisbon), Jakob Henriksson, who got me over my dislike of software engineering, Thomas Eiter and Roman Schindlauer, who have not only made my life writing REWERSE reports so very easy but also have constantly been inspiring me with their research, Massimo Marchiori, who will always be the face

of Venice for me, Jan Małuszyński, whose encouragement, patience, and unending energy amazes me, Pierro Bonatti, Daniel Olmedilla, and Stefano Bertolo, who always challenged us with new perspectives and insight into our work and its relations in REWERSE and outside, to name just a few.

Even when having fun, there are times one looks for motivation and inspiration. One of the persons whose style and quality of research have been a constant inspiration is, starting in the days working on XPath with Dan, Georg Gottlob who has also been so kind to review and help improve this thesis.

Though it took me some time to understand that, a great personal and professional inspiration for me has also been my brother, Filipp Furche, who has, in his own, subtle way, always encouraged me to follow the things I could find enjoyment and satisfaction in.

Whenever motivation is in short supply, my mother is willing to lend a supportive ear and to find the right worlds to keep you going. She has also been so kind to review parts of this thesis, as has been Farrol Kahn and Thomas Höhler, who has, for all my time in Munich, been the best supplier of simple, unadulterated fun. Though some of my old friends had to suffer a bit of neglect in the last months, them sticking by me, even when not fully reciprocated, has been truly humbling. Thank you Axel Eilenberger, in particular.

Finally, let me close by expressing my deep-felt gratitude to the support team in the Munich group: Ellen Lilge, Ingeborg von Troschke, Martin Josko, Stefanie Heidmann, and Uta Schwertel. Without Stefanie Heidmann I would still be trying to understand how to properly fill out the hundreds of forms one encounters in a public institution in Germany. Uta Schwertel is not only one of the kindest persons I know, but has done the most amazing job as manager of the REWERSE project, even when facing researchers that stretched every deadline to the utter limit (or beyond).

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

# PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

## ON THE VISION OF VERSATILE WEB QUERY LANGUAGES

**(1)** François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems*, 1(2), 2005.

**(2)** François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. In Amit Sheth and Miltiadis D. Lytras, editors, *Semantic Web-Based Information Systems: State-of-the-Art Applications*, chapter 8. CyberTech Publishing, 2007.

### SURVEYS AND TUTORIALS ON WEB QUERY LANGUAGES:

**(3)** James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In Jan Małuszyński and Norbert Eisinger, editors, *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, number 3564 in Lecture Notes in Computer Science, pages 35–133. Springer, 2005.

**(4)** Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006.

**(5)** James Bailey, François Bry, Tim Furche, Benedikt Linse, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Rich Clients need Rich Interfaces: Query Languages for XML and RDF Access on the Web. In *Proc. of German XML-Tage*, 2006. URL `http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-14`

**(6)** François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-Based Query Answering. In Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola Henze, Paula-Lavinia Pătrânjan, and Robert Tolksdorf, editors, *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, number 3564 in Lecture Notes in Computer Science. Springer, 2007.

## ON XCERPT 2.0

### LANGUAGE SPECIFICATION

**(7)** François Bry, Tim Furche, and Sebastian Schaffert. Initial Draft of a Language Syntax (Xcerpt 2.0 Beta). Deliverable I4-D6, Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics), 2006. URL http://rewerse.net/deliverables/m18/i4-d6.pdf[1]

**(8)** Tim Furche, François Bry, and Sebastian Schaffert. Xcerpt 2.0: Specification of the (Core) Language Syntax. Deliverable I4-D12, Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics), 2007. URL http://rewerse.net/deliverables/m36/i4-d12.pdf[1]

### VERSATILITY AND RDF ACCESS IN XCERPT

**(9)** Tim Furche, François Bry, and Oliver Bolzer. XML Perspectives on RDF Querying: Towards integrated Access to Data and Metadata on the Web. In *Proc. GI-Workshop on Grundlagen von Datenbanken*, pages 43–47, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-13

**(10)** Tim Furche, François Bry, and Oliver Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 3703 of *Lecture Notes in Computer Science*, pages 72–84. Springer, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-38

1 All REWERSE deliverables have been peer reviewed both within the research project REWERSE and by the external reviewing panel appointed by the EU.

**(11)** François Bry, Tim Furche, and Benedikt Linse. Let's Mix It: Versatile Access to Web Data in Xcerpt. In *Proc. of Workshop on Information Integration on the Web (IIWeb)*, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-16

**(12)** François Bry, Tim Furche, and Benedikt Linse. Data Model and Query Constructs for Versatile Web Query Languages: State-of-the-Art and Challenges for Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 90–104, 2006

**(13)** François Bry, Tim Furche, Clemens Ley, and Benedikt Linse. RDFLog: Filling in the Blanks in RDF Querying. Technical Report PMS-FB-2008-01, University of Munich, 2007. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2008-01

MODULES FOR XCERPT

**(14)** Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Paula-Lavinia Pătrânjan. A Generic Module System for Web Rule Languages: Divide and Rule. In *Proc. Int'l. RuleML Symp. on Rule Interchange and Applications*, 2007. URL http://www.pms.ifi.lmu.de/publikationen/.

**(15)** Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Jendrik Johannes. Modular Web Queries—From Rules to Stores. In *Proc. Int'l. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2007. URL http://www.pms.ifi.lmu.de/publikationen/.

QUERYING AND UPDATE:

**(16)** François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 3208 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2004.

XCERPT SYSTEM DEMONSTRATIONS:

**(17)** Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: Twin Query

Languages for the Semantic Web. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-23.

**(18)** Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Querying the Standard and Semantic Web using Xcerpt and visXcerpt. In *Proc. European Semantic Web Conf. (ESWC)*, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-16.

**(19)** Sacha Berger, François Bry, and Tim Furche. Xcerpt and visXcerpt: Integrating Web Querying. In *Informal Proc. ACM SIGPLAN Workshop on Programming Language Technologies for XML (Plan-X)*, page 84, 2006.

**(20)** Sacha Berger, François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Effective and Efficient Data Access in the Versatile Web Query Language Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 219–224, 2006

**(21)** Sacha Berger, François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Beyond XML and RDF: The Versatile Web Query Language Xcerpt. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 1053–1054, 2006.

CASE STUDIES:

**(22)** Andreas Doms, Tim Furche, Albert Burger, and Michael Schroeder. How to Query the GeneOntology. In *Symposium on Knowledge Representation in Bioinformatics (KRBIO)*, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-15.

**(23)** Loïc Royer, Benedikt Linse, Thomas Wächter, Tim Furche, François Bry, and Michael Schroeder. Querying the Semantic Web: A Case Study. In *Revolutionizing Knowledge Discovery in the Life Sciences*. Springer, 2006

**(24)** François Bry, Tim Furche, Alina Hang, and Benedikt Linse. GRDDLing with Xcerpt: Learn one, get one free! In *Proc. European Semantic Web Conf. (ESWC)*, 2007

XPATH AND XPATH REWRITING:

**(25)** Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *Lecture Notes in Computer Science*. Springer, 2002. [2]

XCERPT OPTIMIZATION:

**(26)** Sacha Berger, François Bry, Tim Furche, and Andreas J. Häusler. Completing Queries: Rewriting of Incomplete Web Queries under Schema Constraints. In Massimo Marchiori, Jeff Z. Pan, and Christian de Sainte Marie, editors, *Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR)*, 2007.

## EVALUATION OF WEB QUERY LANGUAGES

STREAMED EVALUATION OF XPATH QUERIES:

**(27)** François Bry, Tim Furche, and Dan Olteanu. Datenströme. *Informatik Spektrum*, 27(2), 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-2

**(28)** Dan Olteanu, Tim Furche, and François Bry. Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In *Proc. British National Conf. on Databases (BNCOD)*, pages 31–44, 2003. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2003-15.

**(29)** Dan Olteanu, Tim Furche, and François Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *Data Streams Track,Proc. ACM Symp. on Applied Computing (SAC)*, pages 627–631, 2004.

**2** This article is the result of prior work but closely related to and of significant impact on the issues presented in this thesis.

**(30)** François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In *Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM)*. ACM Press, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-32.

## IMPLEMENTATION OF WEB QUERY ENGINES

**(31)** François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The XML Stream Query Processor SPEX. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 1120–1121, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-1.

**(32)** François Bry, Tim Furche, and Benedikt Linse. AMachoS - Abstract Machine for Xcerpt: Architecture and Principles. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 105–119, 2006

...

# CONTENTS

# 1 INTRODUCTION



When we reflect about the ingredients of today's success of the World-Wide Web, one ingredient seems to be essential: easy access to information from diverse sources, each publishing without central control. The enabling technology has been, arguably, the use of a single universal representation format, HTML and its variants. This allows easy access requiring only some form of HTML browser: everyone can collect and aggregate information, e.g., for indexing such as Google does, or for extracting personal information as Spam-Bots do. Surprisingly, this picture changes, when we move to the Web 2.0, the Semantic Web, or whatever other vision of the next generation Web currently en vogue. For these visions we happily build islands: an XML island, an RDF island, a JSON island, an OWL island, a Topic maps island, etc. Of course, there is often legitimate reason to use different representation formats for different kinds of data on the Web. Furthermore, significant resources have been committed to the deployment of various XML and, to a lesser extent, RDF islands.

Unless we assume that all information nicely fits into one and only one of these islands, we have to consider that, increasingly, Web applications will not only process HTML, but also XML, JSON, RDF, OWL, etc. This is already true for most Web 2.0 applications. When we build such applications, however, we do not want to care about the actual data formats but focus on the task of the application.

In this work, we present a solution to this challenge. We argue that query languages such as XSTL, XQuery, SPARQL, XPath, or Xcerpt, which have seen significant success for accessing each of the data islands, should make it easier for the user to access data in different formats. We call for such *versatile* query languages as tools to bridge the data islands and to allow the integration of data in XML, RDF, Topic Maps, or whatever other format. Though choosing a different solution, the W3C has recognized the importance of such scenarios in the recent GRDDL [77] standard, an approach to bring XML data and microformats to the RDF island.

Naturally, we do not stop at the call for such query languages but use that only as the starting point to cast a novel perspective on Web querying, its formal foundation and evaluation. Altogether, this work is divided in three parts:

## 1.1 VISION AND EXEMPLAR: XCERPT 2.0

First, (Part I) we elaborate on the vision of versatile query languages and discuss how that vision has guided the refinement of Xcerpt [188, 187] towards Xcerpt 2.0. Furthermore, we illustrate the use of Xcerpt for use cases developed by the W3C in the context of GRDDL and contrast this approach with the use of separate query languages for XML and RDF as proposed by the W3C. This highlights the advantages of versatile query languages when integrating data from diverse sources with different data formats.

## 1.2 COMMON FORMAL FOUNDATION: CIQLOG

Second, we introduce (Part II) as a formal foundation for Web queries against any data format, CIQLOG, a rule-based query language tailored to semi-structured queries. CIQLOG is a slightly modified variant of $\text{datalog}^{\neg}_{new}$, i.e., *datalog extended with negation and value invention*, or ILOG [132]. CIQLOG is used as a tool to map most major Web query languages into the same formal framework in Part III. Specifically, the considered languages are XPath, XQuery, Xcerpt, and SPARQL.

The common data model and query language CIQLOG and the translations from these languages into CIQLOG yield

**(1)** a *purely logical semantics* for XPath, XQuery, Xcerpt, and SPARQL. For XQuery and SPARQL, this is the first purely logical semantics, to the best of our knowledge.

**(2)** a better *understanding of commonalities and differences* between these languages. In particular, the step from XPath to (composition-free) XQuery illustrates how only a small number of additional features dramatically affects the semantics and evaluation of XQuery.

**(3)** together with the CIQCAG algebra (and the translation from CIQLOG to CIQCAG in Chapter 13),

**(a)** a space optimal *implementation of navigational XPath* with space complexity $\mathcal{O}(q \cdot d)$ and time complexity $\mathcal{O}(q \cdot n)$ where $q$ is the query size, $n$ the data size, and $d$ the depth of the tree data.

**(b)** the *first implementation for SPARQL* with *polynomial-time complexity for tree queries* on arbitrary graphs and linear complexity on tree and certain graph data (see below).

**(c)** efficient implementations for Xcerpt and XQuery that scale over different data and query shapes, i.e., that provide on each restricted class time and space complexity rivaling the best known approaches limited to that class.

**(4)** a foundation for the *integration of queries* from several of these languages (with additional CIQLog interface rules to properly expose data from queries in one language to queries of another language).

## 1.3 EVALUATION: CIQCAG ALGEBRA

Third, we consider the evaluation of CIQLog queries and versatile Web query languages focused on the following question: What is the cost of the move from specialized Web query languages to versatile ones comes, considering time or space complexity?

We assign meaning to things by enumerating their features (or properties or attributes) and placing them in relation with other things. This enables us to distinguish, classify, and, eventually, act upon such things. The same applies to digital data items: to find, analyse, classify, and, eventually, use as basis for actions we need to place data items in relation to other data items: A book to its author, a bank transaction to the bank, the source and the target of the transaction, a patient to its treatment history, its doctor, etc.

How we describe these relations between data items (as well as their features) is the purview of data models. Recently the relational data model, tailored to relations of arbitrary shape, has been complemented by semi-structured data models tailored to Web data. What sets these data models apart from relational data is an even greater focus on relations or links while delegating features or attributes to second-class citizens or dropping them entirely, as in RDF. At the same time most of these data models share a strong hierarchical bias: XML is most often considered tree data.[1] RDF and other ontology languages allow arbitrary graphs but ontologies often have dominantly hierarchical "backbones", formed, e.g., by subclass or part-of relations. To summarize, *structure* is a central property of data and data models determine what shape those structures may take.

1 Though ID-links justify a more graph-like view of XML. Similarly, many XML applications add non-tree linking mechanisms such as HTML hypertext links.

When we want to actually *do* something with the data represented in any of these data models, we use queries. Again, exploiting the relations among sought-for data items is essential: to find all authors of books on a given topic, to find the bank with the highest transaction count, to identify an illness by analysing patterns in a patient's health records. Thus queries mirror the structure of the sought-for data, though often with a richer vocabulary, allowing, e.g., for recursive relation traversal or don't-care parts: a patient is chronically ill if there is some illness (we don't care which illness) that recurs regularly in that patient's health records. Queries may contain additional derived (i.e., not explicitly represented or "extensional") relations—such as equalities or order between the value of data items. The shape of a query is, thus, not limited to the shape of the data but may contain additional relations. To summarize, as for data, structure is a central feature of queries and mirrors the structure of the sought-for data. However, the structure of a query is linked to the structure of the query only if we consider exclusively extensional relations in the query.

The reason we should care about the shape of data or queries is a growing canon of approaches that obtain better complexity and performance for query evaluation if certain limits are imposed on the shape of data, queries, or both.

If we consider arbitrary data, we have little reliable means for compacting relation information. On tree data, in contrast, we can use any number of encodings, e.g., interval encodings [86, 85], hierarchical or path-based labeling [173], or schemes based on structural summaries [200]. In essence, these encodings exploit the observation that structural relations in ordered trees follow certain rules, e.g., each node has a unique parent, the descendants of each node are contained in the descendants of all its ancestors, each node has a unique following and preceeding sibling, etc. Interval encodings on trees, e.g., allow us to compact closure relations quadratic in the tree size into a linear size interval encoding.

For queries, we can make a similar observation: if we allow arbitrary "links" in a query, we need to manage relations between bindings for all nodes in the query at once. However, the relations between the nodes may be limited, e.g., if the *query* is tree shaped (and this holds also for graph-shaped *data*), bindings for each node are directly related only to bindings of its "parent". In fact, if we consider the answers to a query as a relation with the nodes as columns, answers of a tree-shaped query always exhibit multivalued dependencies [91]: In fact, we can normalize or decompose such a relation for a query with $n$ nodes into $n - 1$ separate relations that together faithfully represent the original relation (*lossless-join* decomposition to binary relations over each pair of adjacent variables in the query). This allows us to compact an otherwise potentially exponential

answer (in the data size) into a polynomial representation.

Neither observation is particularly new: acyclic or tree queries on relational data as interesting polynomial-time subclass have been studied, e.g., in [203] and [110]. More recently, the increasing popularity of Web data such as XML triggered renewed interest and reinvestigation of tree data and tree queries as interesting restrictions of general relational structures and queries. Several novel techniques tailored to XML data and XPath or similar tree queries have shown the benefit of exploiting the hierarchical nature of the data for efficient query evaluation: polynomial twig joins [48]; XPath evaluation [113]; polynomial evaluation of tree queries against XML streams [9, 171, 168], linear tree labeling schemes [119, 200] allowing constant time access to structural closure relations such as descendant or following; and path indices [71] enabling constant time evaluation of path queries.

As stated, these techniques have received considerable attention when data and queries are tree shaped. However, data often contains some non-tree aspects, even if the tree aspects are dominant, e.g., ID-links in XML or many ontologies (such as the GeneOntology [87]). Practical queries (such as XQuery or Xcerpt) often go beyond tree queries, e.g., to express value or identity joins, even if they contain a majority of structural conditions. Driven by such considerations, interest in adapting above techniques beyond trees is growing (e.g., labeling and reachability for graph data [199, 195] or hypertree decomposition for polynomial queries beyond trees [108]).

Therefore, we explore in this work means of building on the above mentioned techniques but pushing them beyond trees. We orient this exploration along the following two questions:

**(1)** Can we find an interesting and practically relevant class of (data) graphs that is a proper superset of trees, yet to which algorithms such as twig joins [48], so far limited to trees or DAGs [70], can be extended without affecting (time and space) complexity?

**(2)** Can we integrate the above technologies in the processing of general graph queries in such a way that (often significant) hierarchical components can be evaluated using polynomial algorithms, limiting the degradation of query complexity to non-tree parts of the query? This is particularly desirable as we observe that data and, particularly, queries are often mostly trees with only limited non-tree parts, but *any* non-tree part renders most of the techniques discussed above for tree queries inapplicable.

In the following, we answer both questions essentially positive by introducing a novel algebra, called CIQCAG, the **c**ompositional, **i**nterval-based **q**uery and **c**omposition **a**lgebra for **g**raphs. CIQCAG is a fully algebraic approach to querying Web data (be it in XML, RDF, or other semi-structured

shape) that is build around two central contributions:

**(1)** a novel *characterization of (data) graphs* admissible to interval-based compaction. For this new class of data, called **continuous-image graphs** (or CIGs for short), we can provide linear-space and almost linear-time algorithms for evaluating tree queries rivaling the best known algorithms for tree data.

**(2)** An algebra for a two-phase evaluation of queries separating a tree core of the query (evaluated in the first phase) from the remaining non-tree constraints (evaluated in a second phase). The operations of the algebra closely mirror relational algebra (and can, in fact, be implemented in standard SQL), but **(a)** a novel *data structure* influenced by Xcerpt's memoization matrix [187, 52] and the complete answer aggregates approach [161] allows for exponentially more succinct storage of intermediate results for the tree core of a query. Together with a set of operators on this data structure, called **sequence map**, this enables CIQCAG to **(b)** evaluate almost-tree queries with nearly polynomial time limiting the degradation in performance to non-tree parts of the query. **(c)** Finally, the algebra is tailored to be agnostic of the actual realization of the used relations. This makes it particularly easy to integrate approaches for *arbitrary derived relations* and *indices* in addition to extensional relations, reachability indices such as interval labeling [116] for tree data or [195] for graph data and path indices such as DataGuides [107] or [71] for tree data.

Part I

# USE CASES. VERSATILE WEB QUERYING

# 2 VERSATILE WEB QUERIES—THE VISION



This chapter is a based closely on [58] and [61] with slight refinements and a preview of how the discussed principles are realized in this work.

## 2.1 INTRODUCTION

We present in this work a novel perspective on Web query languages that starts with the vision of a versatile Web query language presented in this chapter. At its core, this vision calls for languages capable of integrating access to diverse Web formats. We show how this vision has guided the development and refinement of Xcerpt 2.0 in Chapter 3. We illustrate how that refinement can be used to implement a versatile query scenario proposed by the W3C, the querying of XML micro-formats through RDF views by means of GRDDL in Chapter 4.

*From format to language versatility*

However, in this work we go beyond the single-language perspective of this chapter and introduce in Part II a formal foundation for Web queries that is expressive enough to capture diverse languages such as Xcerpt, XPath, XQuery, and SPARQL as demonstrated in Part III. Though that

framework is expressive, it still exhibits efficient evaluation of different kinds of Web queries such that, as shown in Part IV, the evaluation of tree queries on tree data is space optimal and rivals the time complexity of the best known previous approaches. At the same time, the approach can also handle tree queries on graph data, often even with the same complexity as on tree data (if the graph is a continuous-image graph). Even graph queries are handled and performance degeneration is limited to the non-tree portion of the query.

We demonstrate by this that versatile query languages as proposed in this chapter can be implemented just as query languages specialized to the existing Web formats. Furthermore, we show that even if there is no versatility on the language level, e.g., if we employ XQuery for XML access and SPARQL for RDF access, those queries can be realized in the proposed formal framework (and thus, e.g., profit from query optimization beyond the borders of each language) and, given a suitable but, compared to a full versatile query language, rather small integration interface, even access results of each other. Such interfaces can, e.g., be SPARQL XML result formats [19] or SPARQL annotations[1].

Nevertheless, all this does not alleviate the effort for the query programmer to handle different query paradigms, e.g., XQuery and SPARQL, in addition to different data formats. We believe that an integration within a language, as argued in the following, is preferable to integration between languages.

*Why versatility matters . . .*

After a decade of experience with research proposals as well as standardized query languages for the conventional Web and following the recent emergence of query languages for the Semantic Web a reconsideration of design principles for Web and Semantic Web query languages is called for.

The "Semantic Web" is an endeavor widely publicized in 2001 by an influential but also controversial article from Tim Berners-Lee, James Hendler, and Ora Lassila [33]. The "Semantic Web" vision is that of the current Web which consists of (X)HTML and documents in other XML formats extended by metadata specifying the meaning of these documents in forms usable by both human beings and computers.

One might see the Semantic Web metadata added to today's Web documents as "semantic indices" similar to encyclopedias. A considerable advantage over paper-printed encyclopedias is that the relationships expressed by Semantic Web metadata can be followed by computers, very much like hyperlinks, and be used for drawing conclusion using automated reasoning methods:

1 http://www.w3.org/2007/01/SPAT/

> *"For the Semantic Web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning." [33]*

A number of formalisms have been proposed in recent years for representing Semantic Web metadata, e.g., RDF [150, 142], Topic Maps [133], and OWL [157]. Whereas RDF and Topic Maps provide merely a syntax for representing assertions on relationships like "a text $T$ is authored by person $P$", schema or ontology languages such as RDFS [45] and OWL allow to state properties of the terms used in such assertions, e.g., that no "person" can be a "text". Building upon descriptions of resources and their schemas (as detailed in the "architectural road map" for the Semantic Web [32]), rules expressed in e.g., SWRL [127] or RuleML [37], allow the specification of actions to be taken, knowledge to be derived, or constraints to be enforced.

Essential for realizing this vision is the integrated access to *all* kinds of data represented in any of these representation formalisms or even in standard Web languages such as (X)HTML, SVG. Considering the large amount and the distributed storage of data already available on the Web, the efficient and convenient access to such data becomes *the* enabling requirement for the Semantic Web vision. It has been recognized that reasonably high-level, declarative query languages are needed for such efficient and convenient access, as they allow to separate the actual data storage from the view of the data a query programmer operates on. This chapter presents a novel position on design principles for guiding the development of query languages that allow access to both standard and Semantic Web data. We believe, it is worthwhile to reconsider principles that have been stated almost a decade ago for query languages such as XML-QL [84] and XQuery [35], then agnostic of the challenges imposed by the emerging Semantic Web.

*Principles of versatility*

Three principles are at the core of the vision of a versatile Web query language:

**(1)** As discussed above, **the same query language should provide convenient and efficient access to any kind of data expected to be found on the Semantic Web**, e.g., to documents written in (X)HTML as well as to RDF descriptions of these documents and even to ontologies. Only by intertwining data from all the different layers of the Semantic Web, that vision can be realized in its full potential.

**(2)** Convenience for the user of the query language requires the reuse of knowledge obtained in another context. Therefore, **the query language should be based upon the principles of referential trans-**

**parency and answer-closedness realized by rules and patterns.** Together, these principles allow (1) for **querying existing and constructing new data by a form-filling approach** (similar to, but arguably more expressive than, the query-by-example paradigm [205, 204]), and (2) for **basic reasoning capabilities** including the provision of different views of the same data, even represented in different Web formalisms.

**(3)** The decentralized and heterogeneous nature of the Web requires **query languages that allow queries and answers to be *incomplete***: In queries, only known parts of the requested information are specified, similar to a form leaving other parts incomplete. Conversely, the answer to a query may leave out uninteresting parts of the matching data.

It is worth noting that the above stated core principles and the more detailed discussion of the design principles in Section 2.2 are describing general principles of query languages, rather than specific issues of an implementation or storage system. Therefore, implementation issues, such as processing model (in-memory vs. database vs. data stream) or distributed query evaluation, are not discussed here. Rather, the language requirements are considered independently of such issues, but allow for further extensions or restrictions of the language, if necessary for a particular setting or application. That the implementation of a versatile query language, though covering access to diverse formats with differing characteristics (e.g., graph vs. tree shape), does not have to suffer performance penalties compared to specialized languages is shown in Part IV where we give an evaluation approach that fits itself to the particularities of the encountered data achieving the same or better complexity for each case as the best known specialized approaches.

These design principles result for a large part from experience in the design of Web query languages by the authors, in particular from the experience in designing the Web query language Xcerpt [188] whose development and refinement has, on the other hand, been influenced by these principles as outlined in Chapter 3. Though Xcerpt is influenced by these principles, it does not (not even in its second incarnation, Xcerpt 2.0) realize all of the below discussed principles. In particular, work on update and evolution languages based on Xcerpt continues as does the investigation of visualization and verbalization. These aspects of a versatile language are not considered further in this thesis.

## 2.2 DESIGN PRINCIPLES

The rest of this chapter is organized around thirteen design principles deemed essential for versatile Web query languages: starting with principles concerning the dual use of a query language for both Web and Semantic Web data (Section 2.2.1) and the specific requirements on how to specify data selection (Section 2.2.2) and the make-up of an answer (Section 2.2.3), further principles regarding declarativity and structuring of query programs (Section 2.2.4) and reasoning support (Section 2.2.5) and finally those regarding the relation of querying and evolution (Section 2.2.6) are outlined.

### 2.2.1 VERSATILITY: DATA, SYNTAX, AND INTERFACE

SINGLE QUERY LANGUAGE FOR STANDARD *AND* SEMANTIC WEB. A hypothesis of this chapter is that a *common query language* for both conventional Web and Semantic Web applications is desirable (this requirement for a Web query language has also been expressed by other authors, e.g., in [166]). There are two reasons for this hypothesis:

*First*, in many cases data is not inherently "conventional Web data" or "Semantic Web data". Instead, it is the usage that gives data a "conventional Web" or "Semantic Web" status. Consider for example a computer science encyclopedia. It can be queried like any other Web document using a Web query language. If its encyclopedia relationships (formalizing expressions such as "see", "see also", "use instead" commonly used in traditional encyclopedia) are marked up, e.g., HTML hypertext links, XLink [83] or any other ad hoc or generic formalism as one might expect from an online encyclopedia, then the encyclopedia can also be used as "Semantic Web data", i.e. as metadata, in retrieving computer science texts (e.g., the encyclopedia could relate a query referring to "Linux" to Web content referring to "operating systems of the 90s") or enhance the rendering of Web contents (e.g. adding hypertext links from some words to their definitions in the encyclopedia).

*Second*, Semantic Web applications will most likely combine and intertwine queries to Web data and to metadata (or Semantic Web data) in all possible manners. There is no reason to assume that Semantic Web applications will rely only on metadata or that querying of conventional Web data and Semantic Web data will take place in two (or several) successive querying phases referring each to data of one single kind. Consider again the computer science encyclopedia example. Instead of one single encyclopedia, one might use several encyclopedias that might be listed in

a (conventional Web) document. Retrieving the encyclopedias requires a conventional Web query. Merging the encyclopedias is likely to call for specific features of a Semantic Web query language. Enhancing the rendering of a conventional Web document using the resulting (merged) encyclopedia is likely to require (a) conventional Web queries (for retrieving conventional Web documents and the addresses of the relevant encyclopedias), (b) Semantic Web queries (for merging the encyclopedias), (c) mixed conventional and Semantic Web queries (for adding hypertext links from words defined in the (merged) encyclopedia).

INTEGRATED VIEW OF STANDARD AND SEMANTIC WEB: GRAPH DATA. Both XML (and semi-structured data in general), as predominantly used on the (standard) Web, and RDF, the envisioned standard for representing Semantic Web data, can be represented in a graph data model. Although XML is often seen as a tree model only (cf. XML Information Set [81] and the XQuery data model [94]), it does provide nonhierarchical relations, e.g., by using ID/IDREF or hypertext links.

Similar to the proposal for an integrated data model and (model-theoretic) Semantics of XML and RDF presented in [176], a query language for both standard and Semantic Web must be able to query any such data in a natural way. In particular, an abstraction of the various linking mechanisms is desirable for easy query formulation: One approach is the automatic dereferencing of ID/IDREF-links in XML data, another the unified treatment of typed relations provided both in RDF and XLink.

The restriction to hierarchical (i.e., acyclic) relations is not realistic beyond the simplest Semantic Web use cases. Even if each relation for itself is acyclic, inference based not only on relations of a single type must be able to cope with cycles. Therefore, a (rooted) graph data model is called for.

THREE SYNTAXES: XML, COMPACT HUMAN-READABLE, AND VISUAL.
It is desirable that a query language for the (conventional and/or Semantic) Web has an XML syntax, to allow easy exchange and manipulation of query programs on the Web. Nevertheless, a second, more compact syntax easier for humans to read and write is desirable. Therefore, two textual syntaxes should be provided: a purely term-oriented XML syntax and another one which combines term expressions with non-term expressions like most programming languages. This other syntax should be more compact than the XML syntax and better readable for human beings. Both syntaxes should be interchangeable (the translation being a low cost process).

Third, a visual syntax can greatly increase the accessibility of the lan-

guage, in particular for non-experts. This visual syntax should be a mere rendering of the textual language, a novel approach to developing a visual language with several advantages: It results in a visual language tightly connected to the textual language, namely it is a rendering of the textual language. This tight connection makes it possible to use both, the visual and the textual language, in the development of applications. Last but not least, a visual query language conceived as a hypertext application is especially accessible for Web and Semantic Web application developers.

MODELING, VERBALIZING, AND VISUALIZING. **(1)** *Authoring and Modeling.* Authoring correct and consistent queries often requires considerable effort from the query programmer. Therefore, semi-automated or fully-automated tool support both for authoring and for reading and understanding queries is essential. **(2)** *Verbalization.* For verbalizing queries, as well as their in- and output, some form of controlled natural language processing is promising and can provide an interface to the query language for untrained users. The importance of such a seemingly free-form, "natural" interface for the Web is demonstrated by the wide-spread success of Web search engines. **(3)** *Visualization.* As discussed above, a visualization based on styling of queries is highly advantageous in a Semantic Web setting. As demonstrated in [28], it can also serve as a foundation for interactive features such as authoring of queries. On this foundation, more advanced authoring tools, e.g., for verification and validation of queries, can be implemented.

### 2.2.2 DATA SELECTION: PATTERN-BASED, INCOMPLETE

Every query language has to define means for accessing or selecting data. This section discusses principles for data selection in a Web context.

PATTERN QUERIES. Patterns (as used, e.g., in Xcerpt [188] and XML-QL [84]) provide an expressive and yet easy-to-use mechanism for specifying the characteristics of data sought for. In contrast to path expressions (as used, e.g., in XPath [73] and languages building upon it), they allow an easy realization of answer-closedness (see below) in the spirit of "query by example" query languages. Query patterns are especially well suited for a visual language because they give queries a structure very close to the structure of possible answers. One might say that query patterns are like forms, answers like form fillings.

INCOMPLETE QUERY SPECIFICATIONS Incomplete queries specify only part of the data to retrieve: e.g. only some of the children of an XML element (referring to the tree representation of XML data called "incompleteness in breadth") or an element at unspecified nesting depth (referring to the tree representation of XML data called "incompleteness in depth"). Such queries are important on the conventional Web because of its heterogeneity: one often knows only part of the structure of the XML documents to retrieve.

Incomplete queries specifying only part of the data to retrieve are also important on the Semantic Web. There are three reasons for this: first, "Semantic Web data" such as RDF or Topic Map data might be found in different (XML) formats that are in general easier to compare in terms of only some salient features. Second, the merging of "Semantic Web data" is often done in terms of components common to distinct data items. Third, most Semantic Web data standards allow data items with optional components. In addition, query languages for the conventional and Semantic Web should ease retrieving only parts of (completely or incompletely specified) data items.

INCOMPLETE DATA SELECTIONS Because Web data is heterogeneous in its structure, one is often interested in "incomplete answers". Two kinds of incomplete answers can be considered. First, one might not be interested in some of the children of an XML (sub-) document retrieved by a query. Second, one might be interested in some child elements if they are available but would accept answers without such elements.

An example of the first case would be a query against a list of students asking for the name of students having an email address but specifying that the email address should not be delivered with the answer.

An example of the second case would be a query against an address book asking for names, email addresses, and if available cellular phone numbers.

But, the limitation of an answer to "interesting" parts of the selected data is helpful not only for XML data. A common desire when querying descriptions of Web sites, documents, or other resources stored in RDF is to query a "description" of a resource, i.e., everything related to the resource helping to understand or identify it. In this case, one might for example want to retrieve only data related by at most $n$ relations to the original resource and also avoid following certain relation types not helpful in identifying a resource.

POLYNOMIAL CORE. The design principles discussed in this document point towards a general-purpose, and due to general recursion most

likely Turing-complete, database programming language. However, it is essential that for the most frequently used queries, small upper bounds on the resources taken to evaluate queries (such as main memory and query evaluation time) can be guaranteed. As a consequence, it is desirable to identify an interesting and useful fragment of a query language for which termination can be guaranteed and which can be evaluated efficiently.

When studying the complexity of database query languages, one distinguishes between at least three complexity measures, data complexity (where the database is considered to be the input and the query is assumed fixed), query complexity (where the database is assumed fixed and the query is the input), and combined complexity, which takes both the database and the query as input and expresses the complexity of query evaluation for the language in terms of the sizes of both [196].

For a given language, query and combined complexity are usually much higher than data complexity. (In most relational query languages, by one exponential factor harder, e.g. in PSPACE vs. LOGSPACE-complete for first-order queries and EXPTIME-complete vs. PTIME-complete for Datalog, cf. [2].) On the other hand, since data sizes are usually much larger than query sizes, the data complexity of a query language is the dominating measure of the hardness of queries.

One complexity class which is usually identified with efficiently solvable problems (or queries) is that of all problems solvable in polynomial time. PTIME queries can still be rather inefficient on large databases. Another, even more desirable class of queries would thus be that of those queries solvable in linear time in the size of the data.

Database theory provides us with a number of negative results on the complexity of query languages that suggest that neither polynomial-time query complexity nor linear-time data complexity are feasible for data-transformation languages that construct complex structures as the result. For example, even conjunctive relational queries are NP-complete with respect to query complexity [69]. Conjunctive queries can only apply selection, projection, and joins to the input data, all features that are among the requirements for query languages for the Semantic Web. There are a number of structural classes of tractable (polynomial-time) conjunctive queries, such as those of so-called "bounded tree-width" [95] or "bounded hypertree-width" [109, 111], but these restrictions are not transparent or easy to grasp by users. Moreover, even if such restrictions are made, general data transformation queries only need very basic features (such as joins or pairing) to produce query results that are of super-linear size compared to the size of the input. That is, just writing the results of such queries is not feasible in linear time.

If one considers more restrictive queries that view data as graphs, or

more precisely, as trees, and only select nodes of these trees, there are a number of positive results. The most important is the one that monadic (i.e., node-selecting) queries in monadic second-order logic on trees are in linear time with respect to data complexity [79] (but have non-elementary query complexity [115]). Reasoning on the Semantic Web naturally happens on graph data, and results for trees remain relevant because many graphs are trees. However, the linear time results already fail if very simple comparisons of data values in the trees are permitted.

Thus, the best we can hope for in a data transformation query language fragment for reasoning on the Semantic Web is PTIME data complexity. This is usually rather easy to achieve in query languages, by controlling the expressiveness of higher-order quantification and of recursion. In particular the latter is relevant in the context of the design principles laid out here. A PTIME upper bound on the data complexity of recursive query languages is achieved by either disallowing recursion or imposing an appropriate monotonicity requirement (such as those which form the basis of PTIME data complexity in standard Datalog or Datalog with inflationary fixpoint semantics [2]).

Finding a large fragment of a database programming language and determining its precise complexity is an important first step. However, even more important than worst-case complexity bounds is the efficiency of query evaluation in practice. This leads to the problem of query optimization. Optimization is usually also best done on restricted query language fragments, in particular if such fragments exhibit alternative algebraic, logical, or game-theoretic characterizations.

### 2.2.3 ANSWERS: ARBITRARY XML, RANKED

ANSWERS AS ARBITRARY XML DATA. XML is the *lingua franca* of data interchange on the Web. As a consequence, answers should be expressible in every possible XML application. This includes both text without mark-up and freely chosen mark-up and structure. This requirement is obvious and widely accepted for conventional Web query languages. Semantic Web query languages, too, should be capable of delivering answers in every possible XML application so as to make it possible for instance to mediate between RDF and XTM (an XML serialization of Topic Maps, cf. [175]) data or to translate RDF data from one RDF syntax into another RDF syntax.

ANSWER RANKING AND TOP-K ANSWERS. In contrast to queries posed to most databases, queries posed to the conventional and Semantic

Web might have a rather unpredictable number of answers. As a consequence, it is often desirable to rank answers according to some application-dependent criteria. Therefore, Web and Semantic Web query languages should offer (a) basic means for specifying ranking criteria and, (b) for efficiency reasons, evaluation methods computing only the top-$k$ answers (i.e., a given number $k$ of best-ranked answers according to a user-specified ranking criterion).

QUERY PROGRAMS: DECLARATIVE, RULE BASED. The following design principles concern the design of query programs beyond the data selection facilities discussed in Section 2.2.2.

REFERENTIAL TRANSPARENCY. This property means that, within a definition scope, all occurrences of an expression have the same value, i.e., denote the same data. Referential transparency is an essential, precisely defined trait of the rather vague notion of "declarativity".

Referential transparency is a typical feature of modern functional programming languages. For example, evaluating the expression `f 5` in the language Haskell will always yield the same value (assuming the same definition of f is used). Contrast with languages like C or Java: the expression `f(5)` might yield different results every time it is called because its definition depends on constantly changing state information.

Referentially transparent programs are easier to understand and therefore easier to develop, maintain, and optimize as referential transparency allows query optimizers to dynamically rearrange the evaluation order of (sub-) expressions, e.g., for evaluating in a "lazy manner" or computing an optimal query evaluation plan. Therefore, referential transparency surely is one of the essential properties a query language for the Web should satisfy.

ANSWER-CLOSEDNESS. We call a query language "answer-closed" if replacing a sub-query in a compound query by a possible (not necessarily actual) single answer always yields a syntactically valid query. Answer-closed query languages ensure in particular that every data item, i.e. every possible answer to some query, is a syntactically valid query. Functional programs can—but are not required to—be answer-closed. Logic programming languages are answer-closed but SQL is not. E.g., the answer person(a) to the Datalog query person(X) is itself a possible query, while the answer "name = 'a' " to the SQL query `SELECT name FROM person` cannot (without significant syntactical changes) be used as a query. Answer-closedness, is the distinguishing property of the "query by example" paradigm [205], even though it is called differently there, separating it from previous approaches

for query languages. Answer-closedness eases the specification of queries because it keeps limited the unavoidable shift in syntax from the data sought for and the query specifying these data.

To illustrate the importance of answer-closedness in the Web context, assume an XML document containing a list of books with titles, authors, and prices (cf. for instance the XML Query Use Case XMP [68]). The XPath [73] query

```
/bib/book/title/text()
```

selects the (text of) titles of books, while a similar query in the (answer-closed) language Xcerpt is

```
bib{{ book{{ title{ var T } }} }}
```

XPath does not allow to substitute, e.g., the string "Data on the Web" for the query, and is thus not answer-closed. In Xcerpt, on the other hand, the following is both an answer to the above query and a perfectly valid query in itself:

```
bib{ book{ title{ "Data on the Web" } } }
```

Answer-closedness is useful, e.g., when joining several documents. For instance, a query could first select book titles in a person's favorite book list and then substitute these titles in the query above:

```
and {
  my-favorite-books {{ title { var T } }},
  bib {{ book {{ title { var T } }} }}
}
```

### 2.2.4 RULE-BASED, CHAINING, AND RECURSION

RULE-BASED. Rules are understood here as means to specify inferred, maybe virtual, data in terms of queries, i.e., what is called "views" in (relational) databases, regardless of whether this data is materialized or not. Views, i.e., rule-defined data, are desirable for both conventional and Semantic Web applications. There are three reasons for this:

First, view definitions or rules are a means for achieving the so-called "separation of concerns" in query programs, i.e., the stepwise specifications of data to retrieve and/or to construct. In other words, rules and view definitions are a means for "procedural abstraction", i.e., rules (view definitions, resp.) are the Prolog and Datalog (SQL, resp.) counterpart of functions and/or procedures.

Second, rules and view definitions give rise to easily specifying inference methods needed, e.g., by Semantic Web applications.

Third, rules and view definitions are means for "data mediation". Data mediation means translating data to a common format from different sources. Data mediation is needed both on today's Web and on the emerging Semantic Web because of their heterogeneity.

BACKWARD AND FORWARD CHAINING. On the Web, backward chaining, i.e., computing answers starting from rule heads, is in general preferable to forward chaining, i.e., computing answers from rule's bodies. While forward chaining is in general considered to be more efficient then backward chaining, there are many situations where backward chaining is necessary, in particular when dealing with Web data. For example, a query might dynamically query Web pages depending on the results of previous queries and thus unknown in advance. Thus, a forward chaining evaluation would require to consider the *whole* Web, which is clearly unfeasible.

RECURSION. On the Web, recursion is needed at least

- for traversing arbitrary-length paths in the data structure,
- for querying on the standard Web when complex transformations are needed,
- for querying on the Semantic Web when inference rules are involved.

Note that a free recursion is often desirable and that recursive traversals of XML document as offered by the recursive computation model of XSLT 1.0 are not sufficient.

SEPARATION OF QUERIES AND CONSTRUCTIONS. Two standard and symmetrical approaches are widespread, as far as query and programming languages for the Web are concerned:

- Queries or programs are embedded in a Web page or Web page skeleton giving the structure of answers or data returned by calls to the programs.
- Parts of a Web page specifying the structure of the data returned to a query or program evaluation are embedded in the queries or programs.

It is a hypothesis of this chapter that both approaches to queries or programs are hard to read (and, therefore, to write and to maintain).

Instead of either approach, a strict separation of queries and "constructions", i.e., expressions specifying the structure of answers, is desirable. With a rule-based language, constructions are rule heads and queries are rule bodies. In order to relate a rule's construction to a rule's query, (logic programming) variables can be employed.

The construction of complex results often requires considerable computation. The separation of querying and construction presented here allows for the separate optimization of both aspects, allowing easier adoption of efficient evaluation techniques.

### 2.2.5 REASONING CAPABILITIES

Versatility (cf. Section 2.2.1) allows access to data in different representation formats, thereby addressing format heterogeneity. However in a Web context, data will often be heterogeneous not only in the chosen representation format but also in terms, structure, etc. Reasoning capabilities offer a means for the query author to deal with heterogeneous data and to infer new data.

*Specific Reasoning as Theories*

SPECIFIC REASONING AS THEORIES. Many practical applications require special forms of reasoning: for instance, efficient equality reasoning is often performed using the so-called paramodulation rule instead of the equality axioms (transitivity, substitution, and symmetry). Also, temporal data might require conversions between different time zones and/or calendar systems that are expressed in a simpler format and more efficiently performed using arithmetic instead of logical axioms. Finally, reasoning with intervals of possible values instead of exact values, e.g., for appointment scheduling, is conveniently expressed and efficiently performed with constraint programming.

For this reason, it is desirable that a query language for the (conventional and Semantic) Web can be extended with so-called "theories" implementing specific forms of reasoning.

Such "theory extensions" can be realized in two manners: **(1)** A theory can be implemented as an extension of the run time system of the query language with additional language constructs for using the extension. **(2)** A theory can be implemented using the query language itself and made available to users of this query language through program libraries. In this case, theories are implemented by rules and queries. Based upon, e.g., the XML syntax of the query language (cf. 2.12) such rule bases can then be queried using the query language itself and maintained and updated by a reactive language such as XChange [53].

QUERYING ONTOLOGIES AND ONTOLOGY-AWARE QUERYING. In a Semantic Web context, ontologies can be used in several alternative ways: First, they can be dealt with by a specialized ontology reasoner (the main disadvantage being the impossibility of adding new domain-specific con-

structs). Second, they can be regarded as descriptions to be used by a set of rules implementing the Semantics of the constructs employed by the ontology. (This is similar to a meta-interpreter and may be slow.) Alternatively, the ontology may be "compiled" to a set of rules.

As discussed in the previous point, the query language should allow for both approaches: extending the query language by specific theory reasoners for a certain ontology language, e.g., OWL-DL, as well as the ability to use rules written in the query language as means for implementing (at least certain aspects) of an ontology language. Examples for such aspects are the transitivity of the subsumption hierarchy represented in many ontologies or the type inference based on domain and range restrictions of properties.

The latter approach is based upon the ability to query the ontology together with the data classified by the ontology. This is possible due to the first design principle. Stated in terms of ontologies, we believe that a query language should be designed in such a way that it can query standard Web data, e.g., an article published on a Web site in some XML document format, meta-data describing such Web data, e.g., resource descriptions in RDF stating author, usage restrictions, relations to other resources, reviews, etc., and the ontology that provides the concepts and their relations for the resource description in RDF.

### 2.2.6 QUERYING AND EVOLUTION

When considering the vision of the Semantic Web, the ability to cope with both quickly evolving and rather static data is crucial. The design principles for a Web query language discussed in the remainder of this section are mostly agnostic of changes in the data: only a "snapshot" of the current data is considered for querying; synchronization and distribution issues are transparent to the query programmer.

In many cases, such an approach is very appropriate and allows the query programmer to concentrate on the correct specification of the query intent. However, there are also a large number of cases where information about changes in the data and the propagation of such and similar events is called for: e.g., in event notification, change detection, and publish-subscribe systems.

For programming the reactive behavior of such systems, one often employs "event-condition-action"- (or ECA-) rules. We believe that the specification of both queries on occurring events (the "event" part of ECA-rules) and on the condition of the data, that should hold for a specific action to be performed, should be closely related to or even embed the general purpose query language whose principles are discussed here (cf.

the reactive language XChange [53] integrating the query language Xcerpt).

## 2.3 RELATED WORK

Although there have been numerous approaches for accessing Web data, few approaches consider the kind of versatility asked for by the vision of versatile query language outlined here. This section briefly discusses how the design principles introduced above relate to selected query languages for XML and RDF data, but does not aim at a full survey over current Web query languages as presented, e.g., in [16] and [100].

VERSATILITY. Most previous approaches to Web query languages beyond format-agnostic information retrieval systems such as search engines have focused on access to one particular kind of data only, e.g., to XML, or RDF data. Therefore such languages fall short of realizing the design principles on versatility described in Section 2.2.1. Connected to the realization that the vision of a "Semantic Web" requires joint access to XML and RDF data, versatility (at least when restricted to these two W3C representation standards) has been increasingly recognized as a desirable if not necessary characteristic of a Web query language, e.g., in [176]. The charter of the W3C working group on RDF Data Access even asks "for RDF data to be accessible within an XML Query context [... and] a way to take a piece of RDF Query abstract syntax and map it into a piece of XML Query" [182].

This recognition, however, has mostly lead to approaches where access to RDF data is added to already established XML query languages: [184] proposes a library of XQuery accessor functions for normalizing RDF/XML and querying the resulting RDF triples. Notably, the functions for normalizing and querying are actually implemented in XQuery. In contrast, TreeHugger [194] provides a set of (external) extension functions for XSLT (1.0) [72]. Both approaches suffer from the lack of expressiveness of the XQuery and XSLT data model when considering RDF data: XQuery and XSLT consider XML data as tree data where references (expressed using ID/IDREF) have to be resolved explicitly, e.g., by a join or a specialized function. Therefore, [184] maps RDF graphs to a flat, triple-like XML structure requiring explicit, value-based joins for graph traversal. TreeHugger maps the RDF graph to an XML tree, thus using the more efficient structural access where possible, however requiring special treatment of RDF graphs that are not tree shaped. None of these approaches fulfills the design principles proposed in Section 2.1 entirely, but they represent important steps in the direction of a versatile Web query language.

DATA SELECTION. For the remainder of the design principles, Web query languages specialized for a certain representation format such as XML or RDF are worth considering. One of the most enlightening views on the state-of-the-art in both XML and RDF query languages is a view considering how data selection is specified in these languages. Both data formats allow structured information and data selection facilities emphasize the selection of data based on its own structure and its position in some context, e.g., an XML document or an RDF graph. For specifying such structural relations, three approaches can be observed:

**(1)** Purely relational, where the structural relations are represented simply as relations, e.g., `child(CONTEXT,X)` ∧ `descendant(X,Y)` for selecting the descendants of a child of some node `CONTEXT`. This style is used in several RDF query languages, e.g., the widely used RDQL [163] and current drafts of the upcoming W3C RDF query language SPARQL [183]. For XML querying, this style has proven convenient for formal considerations of, e.g., expressiveness and complexity of query languages. In actual Web query languages it can be observed only sparsely, e.g., in the Web extraction language Elog [18].

**(2)** Path-based, where the query language allows several structural relations along a path in the tree of graph structure to be expressed without explicit joins, e.g., **`child`**`::*/`**`descendant`**`::*` for selecting the descendants of childs of the context. This style, originating in object-oriented query languages, is used in the most popular XML query languages such as XPath [73], XSLT [72], and XQuery [35], but also in a number of other XML query languages, e.g., in XPathLog [156] that shows that this style of data selection can also be used for data updates. Several ideas to extend this style to RDF query languages have been discussed, but only RQL [138] proposes a full RDF query language using path expressions for data selection.

**(3)** Pattern-based, as discussed in Section 2.2.2. This style is used, e.g., in XML-QL [84] and Xcerpt, but is also well established for relational databases in the form "query-by-example" and datalog.

Most Web query languages consider to some extent incomplete query specifications as Web data is inherently inconsistent and few assumptions about the schema of the data can be guaranteed. However, only few query languages take the two flavors of incomplete data selection discussed in Section 2.2.2 into account (e.g., Xcerpt and SPARQL [183]).

Polynomial cores have been investigated most notably for XPath (and therefore by extension XQuery [35] and XSLT [72]), the results are presented, e.g., in [113, 144, 21, 22].

ANSWERS. Naturally, most XML query languages can construct answers in arbitrary XML. This is, however, not true of RDF query languages, many of which such as RDQL [163] do not even allow the construction of arbitrary RDF, but rather outputs only ($n$-ary) tuples of variable bindings.

Answer ranking and top-$k$ answers have historically rarely been provided by the core of Web query languages, but rather have been added as an extension, see e.g., [10], a W3C initiative on adding full-text search and answer ranking to XPath and XQuery [35]. In relational databases, on the other hand, top-$k$ answers are a very common language feature.

QUERY PROGRAMS. Declarativity and referential transparency have long been acknowledged as important design principles for any query language, as a declaratively specified query is more amenable to optimization while also easing query authoring in many cases.

Most of the Web query languages claim to be declarative languages and, oftentimes, to offer a referentially transparent syntax. In the case of XQuery, the referential transparency of the language is doubtful due to side effects during element construction. For instance, the XQuery

```
let $x = <a/> return $x is $x
```

where **is** is the XQuery node comparator, i.e., tests whether two nodes are identical, evaluates to true, whereas the query `<a/> is <a/>` evaluates to false, although it is obtained from the first query by replacing all occurrences of `$x` with its value.[2] The reason for this behavior lies in the way elements are constructed in XQuery: In the first query a single (empty) `a` is created, which is (of course) identical to itself. However, in the second case, two `a` elements are constructed, which are not identical (and therefore the node identity comparison using **is** fails). Interestingly, this behavior is related to XQuery's violation of design principle 2.4.4, that stipulates that querying and construction should be separated in a query language.

In contrast to referential transparency, answer-closedness can not be observed in many Web query languages. With the exception of Xcerpt, Web query languages provide, if at all, only a limited form of answer-closedness, where only certain answers can also be used as queries.

Related to answer-closedness is the desire to be able to easily recognize the result of a query. This can be achieved by a strict separation of querying and construction, where the construction specifies a kind of form filled with data selected by the query. Such a strict separation is not used in most XML query languages, but can be observed in many RDF query languages,

2 This has been pointed out, in a slight variation, by Dana Florescu on the XML-DEV mailing list, see http://lists.xml.org/archives/xml{}-dev/200412/msg00228.html.

e.g., RDF and SPARQL, due to the restricted form of construction considered in these languages (following a similar syntax as SQL, but restricting the **SELECT** clause to, e.g., lists of variables).

Section 2.2.4 proposes the use of (possibly recursive) rules for separation of concern, view specification. This has been a popular choice for Web query languages (e.g., XSLT [72], Algae[3], in particular when combined with reasoning capabilities (e.g., in TRIPLE [193], XPathLog [156]).

REASONING CAPABILITIES. Reasoning capabilities are, as discussed in Section 2.2.5, very convenient means to handle and enrich heterogeneous Web data. Nevertheless, the number of XML query languages featuring built-in reasoning capabilities is rather limited, examples being XPathLog [156] and Xcerpt. In contrast, several RDF query languages provide at least limited forms of reasoning (e.g., for computing the transitive closure of arbitrary relations), e.g., TRIPLE, Algae. Some RDF query languages also consider ontology-aware querying with RDFS [45] as ontology language. For XML query languages, this has not been considered at length.

## 2.4 EXEMPLARS

In addition to the broad discussion of previous Web query languages by the presented design principles, the following two section illustrates two languages in more detail, viz. XQuery and Xcerpt.

### 2.4.1 CASE STUDY: XQUERY

XQuery is the soon-to-be W3C recommendation for querying XML data. It is characterized by (1) the use of *XPath navigation* for data access, (2) the use of *FLWOR expressions, a form of iterations*, to iterate over the result of query expressions, and (3) a *mix of construction and querying* allowing for compact, but at times hard to read queries.

In the following, we give a brief overview of XQuery in terms of the design principles from this chapter:

**(1) Versatility:** XQuery has been carefully *tailored to querying XML* data with a data model that closely resembles the XML Information Set [81], which describes the information model of an XML document. Other data representation formats have not been considered during the development, though [Robie et al., 2001] proposes the

3 http://www.w3.org/2004/06/20-rules/

use of XQuery to query RDF data. Following the XML Information Set, XML data is viewed in XQuery as *tree shaped* with no special consideration of ID/IDREF links in the data model. This is part of the reason, why [184] can not make use of XQueries descendant axes for traversal in the RDF graph. *Two syntaxes* for XQuery have been proposed by the W3C, a compact expression syntax that is not an XML format though some of its expressions resemble XML elements and a full XML syntax, called XQueryX, that essentially provides access to the parse tree of an XQuery expression as XML. XQBE [42] is a *visual interface* for a subset of XQuery.

**(2)** **Data Selection:** Instead of pattern- (or example-) based selection of data, XQuery uses *navigation* in XML documents with XPath. XPath is well suited for simple selection tasks, but suffers in the context of a full query language from two weaknesses: First, navigation can become complex due to the multitude of XPath's axes that allow great freedom at navigation in the document, though [170] show that the reverse axes of XPath do not add to the expressive power of XQuery. Together with the challenge to provide efficient evaluation for all of these axes, this has let the developers of XQuery to make most of XPath's axes optional. Second, XPath has been designed as a unary selection language. When multiple related pieces of information are to be selected ($n$-ary queries), XPath expressions must be transformed into full XQuery FLWOR expressions. *Incompleteness of queries* is supported through several XQuery operators, e.g., incompleteness in depth through the descendant axis. *Incomplete answers* can be achieved but require some additional reconstruction of the input. It has been shown in [113] that the complexity of evaluating XPath expressions is polynomial. A larger polynomial core for XQuery is still under investigation.

**(3)** **Answers:** XQuery is able to create answers in arbitrary XML formats. No direct support for top-$k$ answers is provided though the effect can be achieved in some cases using positional predicates. Answer ranking is not considered in the context of XQuery, but a recent proposal for a full-text extension of XQuery also considers ranking, cf. [10].

**(4)** **Query Programs:** XQuery does not use rules, but rather functions to provide a form of separation of concerns. Though referential transparency has been a part of the design goals of XQuery, it is doubtful whether this goal has been reached. Indeed, the main reason for the lack of referential transparency are the side-effects of construction expressions, that is avoided in other languages by a strict separation of querying and construction as argued for in the

design principles. Finally, XQuery expressions are closed under the data model, but fail to be answer-closed due to the use of navigation for data access.

**(5) Reasoning Capabilities:** XQuery does not provide any language constructs for reasoning support.

### 2.4.2 CASE STUDY: XCERPT

Xcerpt [188] is a Web query language developed in the European Network of Excellence REWERSE[4] that aims, in the spirit of the versatility principle, at an intertwined access to different forms of Web data, in particular to RDF and XML. Xcerpt queries are expressed using *patterns* for XML data like in the query-by-example paradigm and *rules* like in logic programming. Xcerpt also adds a number of *novel constructs* to pattern-based XML query languages, e.g., optional query parts and expressive, yet declarative grouping constructs.

In the following, we give a brief overview of Xcerpt in terms of the design principles from this chapter. For more details see Chapter 3.

**(1) Versatility:** Xcerpt has been specifically designed to provide access to different representation formats that can be mapped to a *graph-based* semi-structured data model. Nevertheless, the prime focus of the past development has been access to XML data. Better integration of RDF data is currently investigated. Xcerpt offers a compact, term syntax inspired by logic programming languages and an alternative compact, XML-style syntax that very much resembles XML but annotates elements with additional query constructs. Both these syntaxes are primarily intended for use by humans. An XML syntax similar to XQueryX provides the means of easy programmatic manipulation through XML tools. Finally, visXcerpt [25] provides a *visual interface and editor* based on Web standards such as CSS and ECMA script.

**(2) Data Selection:** Xcerpt uses *patterns* for data selection. Xcerpt's patterns can be *incomplete* in breadth and depth and may contain *optional* or negated parts. Furthermore, *variables* may occur in virtually any place in these patterns making $n$-ary queries, as well as joins easy and concise to express. No results on a polynomial core for Xcerpt have been published, but sub-languages and their computational properties are under investigation.

**(3) Answers:** As XQuery, Xcerpt is able to create answers in arbitrary

4 http://rewerse.net/

XML formats. Top-*k* queries are supported directly using the *first* construct. However, so far only basic arithmetic expressions can be used for ranking. Also no efficient algorithms for evaluating Xcerpt top-*k* queries have been published so far.

(4) **Query Programs:** In constrast to XQuery, Xcerpt uses rules to express views, to mediate data from different sources, to separate reusable parts of a query, and to realize basic reasoning. Xcerpt is both referentially transparent and answer-closed.

(5) **Reasoning Capabilities:** Xcerpt's rules provide a basic reasoning mechanism. Though the current prototypes use backward chaining, a forward chaining evaluation of Xcerpt is conceivable. The integration of domain specific and advanced ontology reasoners into Xcerpt is being investigated, however no results have been published so far.

Xcerpt's versatile features have been further refined in the development of the first major revision of Xcerpt, called Xcerpt 2.0, discussed in the next chapter.

## 2.5 CONCLUSION

In this chapter, design principles for (Semantic) Web query languages have been derived from the experience with previous conventional Web query language proposals from both academia and industry as well as recent Semantic Web query activities.

In contrast to most previous proposals, these design principles are focused on *versatile* query languages, i.e., query languages able to query data in any of the heterogeneous representation formats used in both the standard and the Semantic Web.

As argued in Section 2.4, most previous approaches to Web query languages fail to address the design principles proposed in this chapter, most notably very few consider access to heterogeneous representation formats.

The Web query language Xcerpt [188] that already reflects many of these design principles has been further refined to a true versatile query language along the principles outlined in this chapter. The results are discussed in the following chapter.

We believe that versatile query languages will be essential for providing efficient and effective access to data on the Web of the future: effective as the use of data from different representation formats allows to serve better answers, e.g., by enriching, filtering, or ranking data with metadata available in other representation formats. Efficient as previous approaches suffer from the separation of data access by representation formats requir-

ing either multiple query languages or hard to comprehend and expensive data transformations.

In the following chapters, we use most of these principles to guide the presentation of Xcerpt 2.0 as a versatile Web query language. We illustrate its versatile querying capabilities along a few use cases in Chapters 3 and 4 (that can also serve to better illustrate the principles outlined in this chapter). In Parts II and III, we show that existing Web query languages can be translated into a common, format versatile data model and datalog-like query language. This is exploited in the final parts of the thesis, where we turn to the evaluation of versatile query language and address the concern that versatility comes at too high a price considering complexity and practical performance of query languages (see Part IV).

# 3

# VERSATILE WEB QUERIES WITH XCERPT 2.0—CONSTRUCTS AND EXAMPLES



Section 3.3 is closely based on [51], Section 3.5 on [13] and [12].

## 3.1 INTRODUCTION

The vision and principles of a *versatile* Web query language capable of accessing Web data in different formats, yet not less suitable for each format than a language specialized to that format have guided the continuing development of the Xcerpt [54, 187] query language. Xcerpt as described in [187] has originally been developed with focus on XML data, though viewed as graph data as in the XML Information Set [81] (by resolving ID/IDREF links in the data model). Nevertheless, the design of the language has been, from the beginning, carefully tailored to enable also other Web data formats such as RDF [150] or Topic Maps [133].

This foundation has been exploited to refine Xcerpt, its semantics and formal foundation, and its evaluation towards the vision of versatility outlined in the previous chapter. In this chapter, we focus on refinements to the language Xcerpt itself. In Part II, we describe an integrated formal perspective on Web data models and queries and demonstrate in Chapter 7 how to describe Xcerpt and its data model in terms of that formal framework (an extension of datalog with value invention). Moreover, we extend the framework to include other Web query languages like XPath, XQuery, and SPARQL, thus opening the door for versatility not only on the level of the data but even on the level of the language. In Part IV, we finally show that versatility does not have to come at a price in performance or evaluation complexity: The CIQCAG algebra is proposed that allows the evaluation of Xcerpt, XPath, XQuery, SPARQL, and many other Web query languages but is capable of delivering optimal or at least competitive time and space complexity for restricted cases such as XPath (tree queries on tree data). CIQCAG also extends previous results for (tree) query evaluation on tree data to a substantially larger class of graph data, viz. continuous image graphs. As Chapter 7 connects Xcerpt to our formal foundation for Web data and queries, Chapter 13 connects that framework to CIQCAG by showing how to compile Web queries to CIQCAG expressions.

Returning to the topic of this chapter, we focus, as stated on the refinement of the Xcerpt language towards the vision of versatility. The result is called Xcerpt 2.0 and is, in many ways, a summary of one of the strings of work in the REWERSE working group on "Reasoning-aware Querying". The following chapter presents some of the highlights of that refinement but refrains to address all the technical details (such as the full grammars, the language meta model, etc.). These details can be found in several deliverables and publications of the aforementioned working group, on which this chapter also draws notably: Xcerpt 2.0 is first drafted in [59] and fully specified in [101] which also contains a discussion of node identity and the proper graph data model of Xcerpt 2.0, both REWERSE deliverables. The

versatile aspects of Xcerpt 2.0 are first described in [51]. Finally, the module extensions of Xcerpt 2.0 are based on the modularization framework described in [13] and realized with the REWERSE composition framework Reuseware[1] and has previously been published in [12]. In the following, we first briefly recall Xcerpt 2.0 in Section 3.2, focusing on differences to previous versions of Xcerpt. Then we illustrate the versatility of Xcerpt by a number of examples in Section 3.3. Finally, we highlight two particular issues where the design of Xcerpt 2.0 differs notably from that of previous versions: **(1)** adding node (or object or surrogate) identity to Xcerpt and thus moving from infinite regular trees to graphs as data model in Section 3.4. **(2)** modules with parameters for Xcerpt (and any rule language) in Section 3.5. Though neither of these issues is, in and by itself, novel—XQuery uses node identity, the effect of object identity on query languages in general has also long been investigated, e.g., [1]; regular path expressions have been studied extensively for object-oriented and semi-structure data and are used, e.g., in Lorel [3]; modules for rule languages such as Prolog and Datalog have been considered, e.g., in [47]. However, their application to Xcerpt illustrates the progress of Xcerpt towards flexible, versatile Web queries. In particular, node identity and modules can be considered essential features of a versatile Web query language as node identity is, arguably, necessary to properly support, e.g., occurrence queries in cyclic data or change tracking due to updates. The versatile access to Web data often requires mediating views or rules that give provide an integrated view of data from different sources. Encapsulating such views in modules ensures proper encapsulation of data and separation of concern on the level of tasks or rule sets rather than rules.

## 3.2 XCERPT 2.0: OVERVIEW IN 5000 WORDS

### 3.2.1 XCERPT: A ROUGH SKETCH

Xcerpt is a *semi-structured query language* for the Web, but very much unique among the exemplars of that type of query languages (for an overview see [16] and [100]) in that it combines aspects of different languages in novel ways aiming towards a *versatile query language* as defined in [58, 61] and Chapter 2.

**(1)** In its use of a **graph data model**, it stands closely to early semi-structured query languages such as Lorel [3] than to current W3C

1 http://reuseware.org/

XML query languages such as XPath, XQuery, or XPath. A graph data model enables Xcerpt to faithfully represent ID/IDREF-links in XML as well as arbitrary RDF graphs. Previous versions of Xcerpt [187] lack (node or object) identity and are thus better characterized as having *infinite regular trees* as data model, cf. [80, 1]. However, Xcerpt 2.0 introduces full node identity and identity variables and thus moves towards a graph data model as in Lorel or object-oriented databases. For details see Section 3.4.

**(2)** In its aim to address all specificities of **XML with great care**, it resembles current W3C recommended XML query languages such as XSLT [72] or XQuery [35]. Xcerpt is tailored to XML in numerous ways, e.g., by proper support for attributes, namespaces [44], XML base [151], comments, and processing-instructions. This is achieved without sacrificing the conceptual simplicity and syntactical conciseness of the language. Some aspects of XML are treated differently than in the W3C query languages, e.g., the transparent resolution of non-hierarchical relations.

**(3)** In using (slightly enriched) **patterns** (or templates or examples) of the sought-for data for querying, it resembles the "query-by-example" paradigm [205] and XML query languages such as XML-QL [84]. In contrast, current XPath, XSLT, and XQuery use navigational access to XML data which is very convenient for unary selection where path expressions can be used , but quickly becomes unwieldy for $n$-ary queries where more complex, often nested FLWOR loops must be employed.

**(4)** In offering a **consistent extension of XML** to overcome certain restrictions of XML, that seem arbitrary in the context of Web querying and Xcerpt in particular, it is ready to incorporate access to data represented in richer data representation formats. Instances of such features are siblings whose relative *order is irrelevant* (and can not be queried) and more flexible label alphabets.

**(5)** In providing (syntactical) extensions for querying, among others, RDF, Xcerpt becomes a **versatile query language** (as defined in [58]).

**(6)** In a strict **separation of querying and construction** and in its use of logical variables and deductive rules, it resembles logic programming languages or Datalog. In contrast, SQL and XQuery, e.g., mix construction and querying (nested queries) and use explicit references to views rather than rule chaining.

Most of these characteristics hold also for earlier versions of Xcerpt, but are further strengthened in Xcerpt 2.0. This holds particularly for items 1, 2, and 5 and, in general, strengthens Xcerpt's character as a versatile query language in the sense of Chapter 2.

As briefly mentioned above, Xcerpt uses to a large extent the same concepts for data and queries in that each data item can also serve as a query and a query is mostly an example or pattern of sought-for data. Instead of using separate concepts and syntax for queries (as in navigational query languages such as XQuery [35]), Xcerpt uses terms for representing both data and queries. All data terms are also query terms, but there are some additional constructs in query terms, that allow (a) the extraction of data by using logical variables, (b) the specification of queries that are only incomplete patterns of the data, i.e., where more nodes may occur in the data than specified in the query, and (c) the specification of formulas in terms, i.e., conjunction, disjunction, negation, optionality etc:

- **Logical Variables.** In query terms, logical variables are used to indicate which data is to be selected and to join data (indicated by multiple occurrences of the same variable as in logic programming languages). The result of a query is conceptually a set of tuples each representing a combination of bindings (or matches) for all the variables occurring in the query term. For each tuple, a data term must exist that matches the query where all the variables are substituted by the bindings of the tuple.

- **Separation of Querying and Construction.** In contrast to query languages such as SQL or XQuery, construction and querying are strictly separate in Xcerpt, in particular there are no nested queries in Xcerpt (rather rules and rule chaining is used). The data constructed by a rule is specified in construct terms, that contain variables from the corresponding query terms acting as placeholders for selected data. Additionally construct terms make use of gouping constructs to return all or some of the alternative bindings of a variable.

- **Incomplete Patterns.** In most cases, queries specify just enough restrictions on the data to be returned, as required by the query intent, rather than specifying full or "total" patterns of the data. Xcerpt supports such queries by providing constructs to express that a pattern is incomplete in breadth (i.e., there can be more children than specified), depth (i.e., there can be additional nodes and edges between the matched nodes) etc.

- **Terms as Formulas.** Query terms are not only augmented by variables, but also by constructs for expressing negation, disjunction,

conjunction, and optionality.

In the remainder of this section, we briefly outline Xcerpt's data model and data terms, highlighting the changes in Xcerpt 2.0 compared to previous versions. Then, we discuss how construct and query terms differ from data terms.

### 3.2.2 XCERPT 2.0: DATA MODEL

As stated above, Xcerpt 2.0 uses a GRAPH DATA MODEL. More precisely, Xcert provides access to *one or more* data graphs (that are usually stored in data units called "documents" identified by IRIs [90]). Each data graph is a *rooted, directed, node-labeled, ordered, unranked* graph with two types of nodes:

**Definition 3.1** (Element nodes)**.** Element (or structural) nodes represent XML elements or similar STRUCTURED data items (e.g., resources in RDF) that contain a list of references to further nodes (the node's children).

Each element node is decorated further with a dictionary (or associative list) of (XML-style) ATTRIBUTES. Some attributes are predefined and exist at all nodes, viz. the label and namespace IRI (cf. [44]), others are specified in the data, e.g., as XML attributes. Just like in XML, attributes are single valued and unordered, i.e., for each attribute name (dictionary key) a single value exists and the order of the key-value pairs is not significant and can not be queried. Attributes may be hereditary, i.e., shared by all descendants of a node unless there is an intermediary node that provides a differing value for that attribute. Examples for hereditary attributes are namespaces [44] and base IRIs [151] in XML documents.

In contrast to Xcerpt 1.0, element nodes in Xcerpt have an implicit object or surrogate identity and there are three kinds of equality between element nodes: label (or shallow) equality, structural (or deep) equality, and identity-based equality. The first holds if they have the same label but ignores any child nodes, the second if they have the same label and for each child of one node there is a corresponding child of the other node that are themselves deep equal (in presence of order, the corresponding children must be in the same order), the third only between a node and itself. For details on node identity see Section 3.4.

Element nodes closely resemble ELEMENT INFORMATION ITEMS from [81]. The handling of attributes, however, deviates notably from the XML information set to emphasize the distinction of elements and attributes: attributes are simple key-value pair, where the key is an XML

name (and thus may consist in prefix, IRI, and local name) and the value is an arbitrary string. No further information can be attached to attributes.

Each element node has zero or more edges to other nodes, called its CHILDREN. These edges are always *ordered*. However, in contrast to pure XML, one can specify whether this order is significant, i.e., whether it has to be preserved during storage or transformation and can be queried. All element nodes originating from XML documents are by default ordered. Element nodes where the order is significant are called *ordered*, element nodes where the order is insignificant *unordered*. There are no further restrictions on the edges, i.e., the graph may be cyclic, may have loops, and multi-edges, i.e., the same two nodes may be connected by several nodes, e.g., if a node is the 2nd, 4th, and 12th children of another one.

(XML) element nodes are the ONLY COMPLEX DATA STRUCTURE in Xcerpt. Other complex data structure such as lists (or sequences), homogeneous or heterogeneous records, sets, and dictionaries (or associative lists) can be simulated as terms, but no specific support is offered.

The only other node type is that of atomic or content nodes:

**Definition 3.2** (Content nodes)**.** Content (or atomic) nodes represent data items that are considered UNSTRUCTURED in the context of Xcerpt, i.e., they contain no list of references to further nodes and thus always play the role of leaf nodes in the data graph.

Content nodes can be further distinguished into

**(1)** TEXT NODES that represent the textual content of element nodes. The only attribute of a text node is the string it represents. The same restrictions as for text nodes in XSLT [72], XQuery, and XPath [94] apply, i.e., (1) text nodes *never* represent an *empty string*, (2) two text nodes can *never be direct siblings* of each other. Two nodes are direct siblings, if either they are children of the same ordered element node and are consecutive in the children order or they are children of the same unordered element node. Thus, an unordered element node may not have more than one text node child. If two text nodes are constructed as direct siblings they are *collapsed*.

**(2)** COMMENT NODES that represent comments, i.e., annotations on the actual data that are not meant for machine processing. As text nodes, they have only one attribute: the content of the comment. However, in contrast to text nodes no further restrictions are placed on comment nodes.

**(3)** PROCESSING INSTRUCTION NODES that represent processing instructions, i.e., annotations on the actual data that are meant for

processing by specific "target" services. They carry two attributes, the content of the processing instruction (usually some form of instructions for the "target" service) and the name of the "target" service.

In Chapter 5, the formal notion of CIQLOG data graphs is introduced which are a (slight) generalization of Xcerpt data graphs and a mapping from the Xcerpt data model to CIQLOG data graphs is discussed. This mapping faithfully represents all of the above issues.

### 3.2.3 A SYNTAX FOR DATA: (DATA) TERMS

As syntax for representing data in the above data model, Xcerpt chooses terms. However, these Xcerpt terms extended standard logic terms in several ways to accommodate the richer data model of Xcerpt: We need to add means to represent ordered and unordered terms, cyclic data, attributes, and hereditary information.

An Xcerpt term is called a data term if it maps directly to a data graph as defined in Section 3.2.2. For that, it may only contain four types of terms:

**(1)** ATOMIC OR CONTENT data terms that represent a content node. The most common atomic data term is a simple string representing a text node.

**(2)** STRUCTURED data terms that represents an element node in the data model.

**(3)** REFERENCES to other (structured) data terms expressed by a term identifier.

**(4)** DECLARATION OF HEREDITARY ATTRIBUTES such as namespace or XML base [151] declarations defining a scope for those attributes.

Figure 1 gives an example of an Xcerpt data term drawn from the domain of bibliography management: Mixing typical bibliographic records (similar to Bibtex or DBLP) with actual content (represented as XHTML or in a Docbook-style format) it combines

– so-called document-oriented with data-oriented XML, i.e., data with flexible, recursive structure and data with rather rigid and flat structure. Recursive structure is used, e.g., for the content of articles in Docbook-style format.

```
bib{
 journal.adm @ journal{
  title[ "Applied Data Management" ]
  editors[
   editor-in-chief[ "Titus Pomponius Atticus" ]
   editor(region="Africa")[ "Marcus Aemilius
         Aemilianus" ]
   editor(region="Gaul")[ "Aulus Hirtius"
    affiliation[ "Governor, Transalpine Gaul" ] ]
   editor(region="Cilicia")[ "Marcus Tullius
         Cicero"
    affiliation[ "Governor, Cilicia" ] ]
  ]
  publisher[ "Titus Pomponius Atticus" ]
  volumes[
   journal.adm.v10 @ volume[
    journal.adm.v10.n1 @
          number(type="special-issue"){
     title[ "Data Processing Challenges in the
           Age of Wax Tablets" ]
     editorial[ ^ articles.66.cicero.wax ]
     year[ "60" ]
     month[ "july" ]
    }
    journal.adm.v10.n2 @ number{
     year[ "60" ]
     month[ "november" ]
    }
   ]
  ]
 ]

 conf.dmmc @ proceedings{
  editors[
   editor[ "Marcus Aemilius Lepidus"
    affiliation [ "Consul, SPQR" ] ]
   editor[ "Gaius Julius Caesar Octavianus" ]
   editor[ "Marcus Antonius" ]
  ]
  title[
   "Advancements in Data Management for Military
         and Civil Application"
  ]
  invited-papers[
   ^inproc.44.brutus
   ^article.66.scaurus.qumran
  ]
  abbrev[ "DMMC" ]
  year[ "44" ]
  month[ "july" ]
  location[ "Mutina" ]
  publisher[ "SPQR" ]
 }

 article.66.scaurus.qumran @ article{
  author[ "Marcus Aemilius Scaurus"
   affiliation[ "Tribun, Gnaeus Pompeius Magnus"
         ] ]
  title[ "From Wax Tablets to Papyri: The Qumran
        Case Study" ]
  in(scrolls="102-112")[ ^ journal.adm.v10.n1 ]
  citations [
   cite(ref="article.66.cicero.wax")[ ]
   cite(type="formatted")[ "M. Aemilius Scaurus
         (104): A Case for
       Permanent Storage of Senate Proceedings.
            In: M. Aemilius
       Scaurus, ed. (104): "
       i[ "Princeps Senatus: Honor and
            Responsibility" ]
       ", Chapter 2, 14-88." ]
  ]
 ]

 article.66.cicero.wax @ article{
  authors[
   author[ "Marcus Tullius Cicero"
    affiliation[ "Governor, Cicilia" ] ]
   author[ "Marcus Aemilius Lepidus"
    affiliation[ "Gens Aemilia" ] ]
   author[ "Marcus Tullius Tiro"
    affiliation[ "Secretary, M. T. Cicero" ] ]
  ]
  title[ "Space- and Time-Optimal Data Storage on
       Wax Tablets" ]
  in(scrolls="1-94")[ ^ journal.adm ]
  content(type="xhtml")[
   declare
        ns-default="http://www.w3.org/1999/xhtml"
   body[
    <!-- incomplete due to melted letters on
         tablet -->
    h1(id="contributions")[ "Contributions" ]
    h1[ "A History of Data Storage: From Stone
         to Parchment" ]
    p[ "Despite " cite[ ^
         article.66.scaurus.qumran ] ... ]
    ol[
     li[ em[ strong[ "Homeric" ] " Age:" ] ... ]
     li[ em[ "Age of the " strong[ "Kings" ]
          ":" ] ... ]
    ]
    h1(id="tiro")[ "Notae Tironianae" ]
    img(title="Tironian et" src=...)[ ]
    p[ "As discussed in "
        a(href="#contributions")[ ... ] ]
    h1(id="tachygraphy")[ "Challenges for
         Tachygraphy on Wax" ]
    p[ "Though conditions for writing on wax
         tablets are adverse
      to tachygraphy, systems as in "
         a(href="#tiro")[ ... ] ]
   ]
  ]
 }

 inproc.44.brutus @ inproceedings{
  authors[
   author[ "Marcus Antonius"
    affiliation[ "Consul, SPQR" ] ]
   author[ "Decimus Junius Brutus"
    affiliation[ "Governor, Cisalpine Gaul" ] ]
  ]
  title[ "Efficient Management of Rapidly
       Changing Personal Records" ]
  in(scrolls="24-48")[ ^ conf.dmmc ]
  content(type="docbook") [
   declare ns-default
        "http://example.org/ns/docbook/simplified/1.0"
   section[ info[ title[ "Introduction" ] ]
    section[ info[ title[ "Contributions" ] ]
     para[ "The most notable contributions of
          this article include:"
      list(type="ordered")[
       item[
        para[ "A new " em[ "methodology" ] "
             to ..., cf. "
         pageref(idref="inproc.44.brutus.s1")[
              ... ] ]
        figure[ title[ "Chart of Desertions" ]
             img[ ... ] ]
        para[ "As " cite[
             ^article.66.cicero.wax ] ... ]
       ]
      ]
     ]
    ]
    inproc.44.brutus.s1 @ section[
     info[ title[ "Acknowledgements" ] ]
     para[ "We would like to thank the editors of
          "
      cite[ ^journal.adm.v10.n1 ] ... ]
    ]
   ]
  }
}
```

Figure 1. Exemplary Xcerpt Data Term

- normalized with de-normalized representation of data (e.g., author information is duplicated for each authored paper, whereas the information about the journal is represented once and referenced in other parts),
- hierarchical with delimiter-based structuring of data (e.g., (X)HTML style sections delimited by consecutive h*n* elements vs. nested sections as, e.g., in DocBook),
- resolved and unresolved links. Where links are used to normalize the data (e.g., in case of journal information of a an article), these links are resolved to Xcerpt references. Other links (e.g., the link to another section in the content of an article) are left unresolved as they must be distinguished from "normal" nesting, e.g., links to other sections in the content of an article (like "cf. Section 10").

Structured data terms (line 1, 2, 3, etc.) are distinguished by a label (optionally preceded by a term identifier delimited by `@` and followed by a set of key-value attributes in parentheses) followed by either a curly brace or a square bracket enclosing the children of the term. Curly braces indicate that the order is immaterial, square brackets that it is significant.

Content data terms in this example are string children (line 3, 5, 6, 7, etc.) of structured data terms (enclosed in double quotes) and comments (line 79) denoted as in XML.

References (line 73, 82, etc.) are denoted using `^` and are followed by a term identifier that must be "defined" by placing the same term identifier followed by `@` in front of a structured term (line 2, 29, 50, etc.). The reference is resolved by adding the referred structured term at the place of the reference in the list of children of the reference's parent.

Hereditary attributes are introduced using `declare` blocks (line 107) and hand down their attribute list to all terms in their scope (here the following term and its descendants).

For more details on the syntax of data terms see [101].

### 3.2.4 A SYNTAX FOR QUERIES: (QUERY) TERMS

Query terms specify "patterns" or "examples" of the sought-for data. In fact, any data term is also a query term, but query terms contain three additional aspects beyond data terms: **(1)** *Logical variables* are added to selected nodes, attributes, and labels, **(2)** *Incompleteness* allows queries to specify only relevant portions of a "pattern" stating where matching data terms may contain additional data (and where not). **(3)** *Term formulas* allow the conjunction, disjunction, and negation of query terms.

To better understand these extensions, an intuition of the answer notion in Xcerpt is needed. The questions, which data and construct terms match with a query term, and what the answer (i.e., the substitution multiset) for a query term is, are formally addressed in [187]. At the root of Xcerpt's answer notion stands an extended form of rooted graph simulation (cf. [164, 125] and [104, 88] for more recent work on efficient algorithms for computing simulation and bisimulation). This extension of the classical notion is necessary to accommodate incomplete patterns. In Chapter 7, we discuss an alternative, but equivalent characterization of Xcerpt queries by translation to CIQLog (as introduced in Chapter 6), a variant of datalog with value invention.

Intuitively, a query term without any of the extensions discussed in the following matches only with a data term that has *exactly the same shape modulo reordering of direct sub-terms in unordered structured terms and of attributes in any terms*. The full details are left to [101] and Chapter 7.

VARIABLES. Variables in query terms are used for three purposes: (a) to specify *which parts of a matched (construct or data) term are "selected"* by the query and can be used in the corresponding construct term, (b) to specify *joins*, i.e., multiple occurrences of the same data term or literal value (usually unknown at time of query authoring), and (c) to specify arithmetic or other *conditions* involving (literal) values of variables. Variables may occur in place of structured or content terms, term labels, and attributes. Such variables may occur by itself indicating that they match with any term, label, or attribute that can occur at their position in the pattern or with additional restrictions that limit the shape of the term, label, or attribute and follow the variable separated by →. In Xcerpt 2.0, we also allow variables in place of term identifiers (called then "identity variables" indicated by **idvar**).

INCOMPLETE PATTERNS. Xcerpt introduces a number of concepts to allow query terms to be *incomplete patterns* of the sought-for data, that may specify only what is needed to distinguish relevant from irrelevant data. In contrast to many other query languages (such as XQuery and SQL) that assume that a query specifies only parts of the sought-for data and make it difficult to specify queries where no additional data may occur, Xcerpt patterns make it *obvious* where a query term is incomplete and where not. This is a property that is particularly welcome in the context of semi-structured data as here the schema of the data is often unknown or variable, allowing, e.g., optional or repeated children.

(Structured) query terms can be incomplete with respect to

**(1)** INCOMPLETENESS IN BREADTH, i.e., only a subset of the actual

children of a term is specified. Such a structured query term is called *partial* and indicated by double instead of single braces or brackets. If a term is not *partial*, it is called *total* as discussed above.

Partial terms are obviously essential for dealing with semi-structured data, where the schema of the data may allow for repetition or omissions of data (both in breadth and depth). Thus, queries can not specify total (or complete) patterns for the data.

However, partial terms also introduce a number of new challenges in a pattern language such as Xcerpt—that do not occur, e.g., in logic programming languages such as Prolog, where term arity and children order of two matching terms are always the same.

First, assume an *ordered* query term $q$ that specifies only a partial list of children. Then the position (among its siblings) of a match $m'$ for a child of $m$ may (and will in most cases) differ from the position of $m$ among its siblings (i.e., among $q$'s children). However, in many cases access to the (sibling) position is needed, e.g., to obtain the first child or immediate following sibling of a matched term. Therefore, Xcerpt provides access to the *sibling position* through the position modifier **`position`** $n$.

Second, though one may not be able to specify how all of the children of a sought-for term ought to be shaped like, one might be able to specify how they ought *not* to be shaped. Again, this makes sense only in a partial term, as in a total term the shape of all children must be explicitly stated. Xcerpt uses the **`without`** modifier to express this *subterm negation*.

**(2)** INCOMPLETENESS IN ORDER: As discussed above, data and construct terms may already be distinguished in ordered and unordered terms. Often, however, one might not care about the order in which matches for the sub-terms in a query occur in the data, even if the data itself is ordered. Xcerpt acknowledges this fact by allowing query terms that are unordered to match with ordered terms, but not the other way around. I.e., if the query specifies the order is significant then only data where the order is significant as well can match with that query; if the query however indicates that the order may be ignored, then also data is considered that is ordered, however the sub-terms of the query are matched in any order with the sub-terms of the data.

**(3)** INCOMPLETENESS IN DEPTH: Semi-structured data may not only vary in the number, order, and repetition of children, but also in how elements are nested. E.g., in (X)HTML most inline elements such as `em` may occur in most block-level elements such as `p` or `div`, but may also be nested inside each other. Thus selecting all `em`

elements in an (X)HTML document in a pattern requires a means to specify patterns that are incomplete in depth, i.e., that contain sub-terms that are not direct sub-terms of their parent but stand in another structural relation to it, e.g., occurring at any depth under their parent or occurring at depth 5 under their parent.
To express such incompleteness in depth Xcerpt provides the **desc** modifier, similar in its basic form to the **descendant** axis in XPath. In contrast to XPath (and thus XSLT and XQuery) but as in regular path expressions, Xcerpt 2.0 also provides a more expressive variant of the **desc** modifier that allows direct expression of constraints such as "occurs at depth 5 under its parent" or "occurs at any depth under its parent but with only div elements in between its parent and itself", see [101].

**(4)** OPTIONAL PARTS: One of the most distinguishing features of semi-structured data in contrast to, e.g., relational data aside is the allowance for optional information, i.e., information that occurs in some elements of a certain type but is missing in others of the same type. Though *testing* for the existence or absence of such optional information has been a focus in many semi-structured and XML query languages (most notably structural predicates in XPath), *selecting* of or *construction* based on optional information has been far less closely investigated. Xcerpt provides query authors with a unified concept for handling optional information in the context of testing, selection, and construction, quite in contrast to mainstream XML query languages such as XQuery and XSLT.
Just like in construct terms, the **optional** modifier is used in query terms to indicate which parts of a query may be missing without affecting the matching of the remainder of the query.

**(5)** INCOMPLETE STRING MATCHING: Finally, like in the relational case, queries often may not be able to specify literal content or identifiers completely, but rather query for data where the literal content or the identifiers falls into some class, specified in Xcerpt by means of POSIX.1 regular expressions enhanced with variable bindings: additionally to using POSIX's numeric backreferences, Xcerpt allows subexpressions to be bound to Xcerpt literal variables. This allows the extraction and insertion of data from the rest of the Xcerpt query into the regular expression.

To illustrate these features of Xcerpt query terms consider two example query terms The first selects articles (binding them to the variable X) in a bibliography as in Figure 1, but only if they contain at any depth an author element with only content "Cicero" (the "only" is due to the single braces for the author which indicate that this is a total query term and thus that

there may be no additional children in a matching data term). Due to the **desc** authors at any depth are matched and thus also articles that cite articles with author "Cicero" are matched:

```
bib{{
  var X → article{{
    desc author{
      "Cicero"
    }
  }} }}
```

The second query term is a slight variant where we show the effect of multiple occurrences of the same variable: It selects again authors but this time only if they have the same author as some other article (and the author is a direct child due to the absence of a **desc**).

```
bib{{
  var X → article{{
    author{
      var Y
    } }}
  article{{ author{ var Y } }} }}
```

The final example illustrates the difference between normal and identity variables as discussed in Section 3.4: It selects articles that are cited by at least two different articles. Optionally, it also selects their citations list, but the absence of such a list does not preclude the parent article to be included in the bindings for X.

```
bib{{
  article{{ cite{{
            var Article → idvar cited @ article{{
              optional var Citation → citations{{ }}
            }} }} }}
  article{{ cite{{ idvar cited @ article{{ }} }} }}
```

For a more detailed discussion of query terms and their matching see Section 7.3.

### 3.2.5 A SYNTAX FOR RESULTS: (CONSTRUCT) TERMS

Given the bindings for variables in a query term, construct terms are used to create new data (terms) based on these bindings and the specification of the resulting data given by the construct term itself.

As such a construct term is essentially a data term extended with variables, as in query terms, in place of structured terms, labels, or attributes.

We also allow identity variables in construct terms indicating reference instead of copy semantics for that term.

There are two major differences between Xcerpt result construction (and similar query languages for structured data) when compared to fixed-arity Datalog or Prolog: **(1)** Given, for instance, bindings for authors and titles, one would like to create author elements that contain for each author all (arbitrarily many) corresponding titles. Explicit support for *grouping constructs* in the scope of other data is needed to express that form of construction. **(2)** Given, for instance, bindings for authors and titles, one would like to create author elements that contain the name of the author and, if available, its affiliation and all other authors with the same affiliation wrapped in a related element. Since, the data does not contain affiliations for all authors, we use **`optional`** query terms to select that information. Correspondingly, we use conditional or optional construction in the head such that only if there is a matching affiliation for an author the related element and its content is created.

Consider, for example, the following substitutions for the variables `Author`, `Title`, and `Publication` given as result of a query:

| `Author` | `Title` | `Publication` |
|---|---|---|
| "Cicero" | "Data Processing ..." | **`null`** |
| "Cicero" | "Space and ..." | journal[2] |
| "Antonius" | "Advancements ..." | **`null`** |
| "Antonius" | "Efficient Manage..." | proceedings[29] |
| "Tiro" | "Space and ..." | journal[2] |

Notice, that `Publication` is an optional variable.

Then the following construct term

```
result[
  all author[ var Author ] ]
```

results in the single data term

```
result[
  author[ "Cicero" ]
  author[ "Antonius" ]
  author[ "Tiro" ] ]
```

Notice, how Xcerpt defaults to grouping by structural equivalence and thus treats the two substitutions with author "Cicero" as one group, constructing only a single result data term for them.

If we add `Title` as free variable in the scope of the grouping modifier, the grouping variables and thus the groups change:

```
result[
 all author[
      var Author
      title[ var Title ]
 ] ]
```

Leading to the result:

```
result[
 author[ "Cicero"
   title[ "Data Processing ..." ] ]
 author[ "Cicero"
   title[ "Space and ..." ] ]
 author[ "Antonius"
   title[ "Advancements ..." ] ]
 author[ "Antonius"
   title[ "Efficient Manage..." ] ]
 author[ "Tiro"
   title[ "Space and ..." ] ]
]
```

Now the substitutions for author and title are both considered for forming a group, leading to more groups!

Nesting grouping modifiers also affects the free variables, e.g., in the following construct term `Title` is no longer free for the out **`all`** only for the inner.

```
result[
 all author[
      var Author
      all title[ var Title ]
    ]
]
```

Thus the result on the sample substitutions is:

```
result[
 author[ "Cicero"
   title[ "Data Processing ..." ]
   title[ "Space and ..." ]
 ]
 author[ "Antonius"
   title[ "Advancements ..." ]
   title[ "Efficient Manage..." ]
 ]
 author[ "Tiro"
```

```
    title[ "Space and ..." ] ]
12 ]
```

Combining grouping an optional modifiers can lead to surprisingly expressive constructs:

```
result[
2  all author[
     var Author
4    all (title[ var Title ]
         optional var Publication
6            with-default standalone[ ])
    ]
8 ]
```

Results in the following data term:

```
result[
2  author[ "Cicero"
    title[ "Data Processing ..." ]
4   standalone[ ]
   ]
6  author[ "Cicero"
    title[ "Space and ..." ]
8   journal.adm @ journal[ ... ]
   ]
10 author[ "Antonius"
    title[ "Advancements ..." ]
12  standalone[ ]
   ]
14 author[ "Antonius"
    title[ "Efficient Manage..." ]
16  conf.dmmc @ proceedings[ ... ]
   ]
18 author[ "Tiro"
    title[ "Space and ..." ]
20  journal.adm @ journal[ ... ]
   ]
22 ]
```

Again, the full details of construct terms are discussed in [101] and further examples of construct terms and their instantiation can be found in Section 7.3.

### 3.2.6 A SYNTAX FOR PROGRAMS: RULES

Query and construct terms are combined to form rules with query terms as body and construct terms as head. The body of rules may also be formulas over query terms.

The following example shows an Xcerpt rule that combines one of the above example query terms with a simple construct term that puts all the selected articles "on a shelf":

```
CONSTRUCT
  shelf{ all var X }
FROM
  bib{{
    var X → article{{
      author{
        var Y
      } }}
    article{{ author{ var Y } }} }}
END
```

Rules can be seen as "views" specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database).

Xcerpt rules may be *chained* like active or deductive database rules to form complex query programs, i.e., rules may query the results of other rules.

This concludes our first look at Xcerpt 2.0. In the following sections, we highlight most of the major additions (or revisions) of Xcerpt 2.0 compared to previous versions. We omit minor revisions to syntax and language structure such as **(1)** the dedicated attribute syntax, **(2)** details of the support for hereditary attributes, and **(3)** term lists in without, grouping, and optional (query or construct) terms which allows Xcerpt to cope with delimiter-based XML data such as XHTML [177]or Apples plist format [11]. These are discussed in more detail in [101]. For Xcerpt 2.0, we have also developed a significantly improved RDF access layer compared to basic RDF support as detailed in [38]. This RDF access layer is described in [180] and largely omitted here.

In Parts II and III, we return several times to Xcerpt and discuss in more detail its data model, queries, and semantics in terms of a translation to CIQLog.

Before we turn to the specific language features, the next section establishes the use of Xcerpt as a versatile query language along a number of examples.

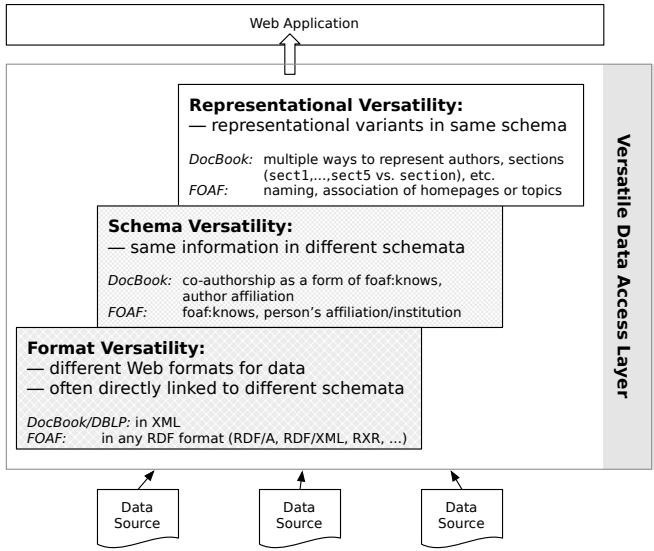



Figure 2. Versatile Data Access on the Web

## 3.3 VERSATILITY 101: VERSATILE QUERIES BY EXAMPLE

Web query languages are tailored to the efficient and effective manipulation of such data. However, conventional Web query languages such as XQuery, XSLT, or SPARQL focus only on one of the different data formats available on the Web and provide little to ease the integration of the heterogeneous schemata or representations. We argue in Chapter 2 that *versatility*, i.e., the ability to handle in the same query program heterogeneous formats, schemata, and representations, is a crucial property of Web query languages, in particular to provide a convenient platform for the development of applications needing integrated access to different Web data sources. As defined in Chapter 2, a language is versatile if it provides means to handle heterogeneous data. One can distinguish three forms of versatility in Web query languages based on the origin of the heterogeneity that is considered by the language (summarized in Figure 2):

**Format versatility:** We call a language *format* versatile if it is able to query data in different (Web) formats such as XML, RDF, and Topic Maps. Whereas in the case of XML there is a precise mapping between serialization and data model, RDF and Topic Maps exhibit many different serialization formats, e.g., for RDF RDF/XML, RDF/A, and RXR. The open and rapidly changing nature of the Web prevents that a small set of "built-in" formats suffices to achieve format versatility. If versatility beyond

mere serialization differences is desired, the different data models have to be taken into consideration, cf. [176].

**Schema versatility:** Schema versatility is a slightly more elusive property of a query language linked to its ability to handle data represented according to different schemata. Ideally, the schema differences are handled transparently, e.g., by integrating the different schemata according to some given schema mapping. Such schema mappings may be specified, e.g., in RDFS, the RDF vocabulary description language, or in OWL, the Web ontology language. Direct or programmed support for schema mappings enables a query formulated according to one of the schemata to be evaluated against data in any of them.

**Representational versatility:** Web data is partially or semi-structured rather than fully structured like relational data. Therefore, even within the same schema, data may be represented in different ways, both w.r.t. structure and datatype. Web query languages should be able to take these representational variants into account, but in contrast to format and schema heterogeneity representational variants can not be handled transparently: For instance, different ways to represent a section in DocBook (`sect1` vs. `section`) carry additional semantics and must be distinguished in some contexts, whereas in many cases they can be considered the same. Such distinctions should be expressible in a versatile query language, allowing the programmer to choose the appropriate solution.

Xcerpt [188, 187] is one of the few Web query languages that address all three forms of versatility. It is a *semi-structured query language*, but very much unique among such languages: (1) In its use of a *graph data model*, it stands more closely to semi-structured query languages like Lorel than to recent mainstream XML query languages. (2) In its aim to address all *specificities of XML*, it resembles more mainstream XML query languages such as XSLT or XQuery. (3) In using (slightly enriched) *patterns* (or templates or examples) of the sought-for data for querying, it resembles more the "query-by-example" paradigm [204] than mainstream XML query languages using navigational access. (4) In its strict separation of querying and construction in rules it allows an easy transformation and interfacing of different rules. (5) In its use of rules as procedural abstraction or view mechanism, it provides a foundation for reasoning and mediation.

Following a short introduction to the data formats considered, the remainder of the article further details the three forms of versatility along concrete examples realized in Xcerpt. Special emphasize is placed on the identification of general principles needed or useful for a versatile Web query language.

### 3.3.1 WEB FORMAT BASICS

Web data is currently, as far as it is not image, video, or layout-centric, mostly represented as semi-structured data, marked up either as XML (most often in the form of XHTML) or in one of the serializations of RDF (most often in the form of RSS). The discussion of versatile data access in Sections 3.3.2–3.3.4 uses data in both formats. To this end, the salient features of the two data representation formats are shortly summarized here.

XML [43] is a generic markup language for semi-structured data that has found widespread adoption both for data exchange and data representation on the Web (and beyond). Its data model is essentially a tree of nodes corresponding to elements (such as `h1` or `title` in HTML) of the XML document. The tree structure reflects the nesting of elements in the serial XML document. Elements may contain text content represented as text node children in the data model. Other features of XML include attributes, namespaces, and processing instructions, for details see [81].

Where XML data is used in the following sections, the examples are mostly drawn from a list of articles, papers, conferences, etc. in the style of DBLP[2], but with additional information about the actual content of the paper in DocBook format[3]. Figure 3 shows a visual representation of parts of the sample XML document (with fictional journal information) using Xcerpt's visual companion language visXcerpt [23].

RDF [150] is the prevalent standard for representing metadata in the (Semantic) Web. RDF data is sets of *triples* or statements of the form (*Subject*, *Property*, *Object*). RDF's data model is a directed graph, whose nodes correspond to subjects and objects of statements and whose arcs correspond to their properties relating subjects and objects. Nodes are labeled by either (1) URIs describing (Web) resources, or (2) literals (i.e., scalar data such as strings or numbers), or (3) are unlabeled, being so-called anonymous or *blank nodes*. Blank nodes are commonly used to group or "aggregate" properties. Edges are always labeled by URIs indicating the type of relation between its subject and object.

*RDFS* allows one to define so-called "RDF Schemata" or ontologies, similar to object-oriented data models. Based on RDFS, inference rules can be specified, for instance the transitivity of the class hierarchy.

2 http://www.informatik.uni-trier.de/~ley/db/

3 http://www.oasis-open.org/docbook/

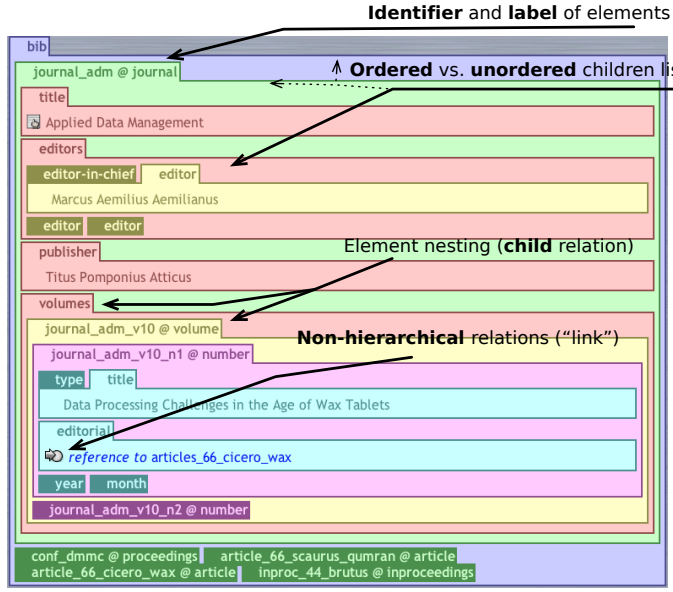



Figure 3. Visual Rendering of Sample XML Data

RDF can be *serialized* in various formats, the most frequent being XML. Early approaches to RDF serialization have raised considerable criticism due to their complexity. As a consequence, a surprisingly large number of RDF serialization have been proposed, cf. [16] for a survey of serialization formats.

In the following, example data based on the "Friend of a Friend" (FOAF) project[4] is used. FOAF is an RDF vocabulary describing mostly foaf:Persons by properties such as foaf:name, foaf:mbox, foaf:homepage, foaf:interest, etc. Furthermore, it allows to establish "social networks" of persons using foaf:knows, foaf:Project, foaf:Group, foaf:Organization, and foaf:member.

### 3.3.2 FORMAT VERSATILITY

The most basic type of versatility a query language for the (Semantic) Web should posses, is *format versatility*. Data on the Web is encountered in many different XML markup languages. In contrast, metadata is usually represented in RDF or Topic Maps. However, XML serializations exist for both of these meta data standards, which makes their integration with XML data easier. Therefore, it has been proposed, e.g., in [184], that an ordinary

4 http://www.foaf-project.org/

XML query language such as XQuery already provides all necessary means to integrate data from these different formats. However, using ordinary XML query languages for such an integration proves to be infeasible for a number of reasons:

(1) *Limitations of the XML Data Model:* The W3C's data model for XML, the XML Infoset [81], deviates from most other semi-structured data models (including those of RDF and Topic Maps) in two notable ways: it is tree-shaped, handling non-hierarchical relations as "second class" relations, and it assumes that the order among the children of a node is always relevant. These limitations are true neither for RDF nor Topic Maps and make format versatile extensions of ordinary XML query languages such as XQuery or XSLT difficult at best: Either the approach has to tackle "slicing" up an RDF or Topic Maps graph in XML trees [197], leading to very unnatural queries, or relational representations are used to represent RDF triples or Topic Maps assertions, e.g., in [184].

(2) *Multitude of Serialization Formats:* Topic Maps and, to an even greater degree, RDF exhibit numerous structured text and XML serialization formats, e.g., the W3C syntax for RDF RDF/XML [20], a syntax for embedding RDF in arbitrary XHTML documents RDF/A [4], and Turtle, the RDF triple syntax adopted for W3C's SPARQL. A format versatile language should thus be able to adopt to rapidly emerging serialization variants.

(3) *Transparent Integration:* Integrating data from different formats with ordinary XML query languages can be quite cumbersome and unnatural, as the integration must be performed as part of each query accessing the integrated data. Xcerpt rules represent a high level construct that can be used to provide a uniform logical view over XML markup languages and RDF serializations.

On the other hand, existing RDF and Topic Maps query languages are equally unsuited for XML processing, partially due to the limited expressiveness of most of these languages (including the W3C's SPARQL), partially due to the specificities of XML related to its document markup origins not considered in these languages.

Therefore, a proper versatile query language is called for that has transparent support for different formats and a data model capable of seamlessly, but without loss of (relevant) information, integrating semi-structured data on the Web represented in either XML, RDF, or Topic Maps.

The remainder of this section illustrates this point along a number of examples realized in Xcerpt. The above mentioned data sources, a DBLP-like article collection and FOAF data are used.

The combination of both of these sources allows, e.g., "to find the phone numbers and Web sites of authors for a given article" (1). Moreover, assum-

ing that co-authors personally know each other, the co-author relation could be inferred from the article collection and used to enhance the existing foaf:knows property (2). Finally, FOAF data might be used to display the images of authors of scientific papers for visitors of the DBLP query interface (3).

There is a plethora of similar use cases, e.g., leveraging data from Amazon combined with interests of persons expressed in FOAF or using Google Maps to display geographical distributions of friends, groups, people with similar interests, etc. Geographical information is a particular rich field of applications where heterogeneous data sources need integration.

Often geographical information is provided in form of XML, e.g., in form of GML (geography markup language) or the, more high-level, CIA factbook information. This could be combined with data from the open directory portal, cf. http://dmoz.org/rdf/. One might be interested in companies that are located in Polish cities with a given minimum population. While the information about the cities is available in XML format in CIA factbook, lists of companies in a city are provided in RDF/XML format by the ODP. A natural difficulty in integrating XML and RDF data is that XML does not intrinsically support the notion of globally unique identifiers as the URI/IRIs in RDF. While the topic *Poland* holds a globally unique identifier in the ODP database (namely http://dmoz.org/rdf/Regional/Europe/Poland), the element that represents Poland in the CIA factbook is not associated with a unique URI. Therefore, joining information about a data item from both sources must depend on properties of the data item not on an unique identifier. An OWL mapping might be used to specify so called inverse functional properties that are like primary keys in relational data, cf. Section 3.3.3.

*From XML to RDF*

Xcerpt is tailored in many aspects closely to XML, making the access to XML data like speaking ones "mother tongue". In this section we show how the co-author relation (indicated by the name dblp:coauthor) can be extracted from the DBLP-like article collection. Consider the following fragment:

```
bib(){
 article.66.cicero.wax @ article(){
  authors()[
   author()[ name(){ "Marcus Tullius Cicero" }
       affiliation()[ "Governor, Cicilia" ] ]
   author()[ name(){ "Marcus Aemilius Lepidus" } ] ]
  title()[ "Data Storage on Wax Tablets" ] }
```

From this XML fragment it can be deduced that there are persons with the names "Marcus Tullius Cicero" and "Marcus Aemilius Lepidus" who are co-author of each other. XML does not associate unique identifiers with the elements in a document. Hence, blank nodes must be used to represent these persons in the corresponding RDF graph. The rules below transform the DBLP-like article collection to a set of RDF-like triples containing all dblp:coauthor predicates for authors of the same paper. For simplicity, it is assumed that names are unique within the article collection. If this assumption does not hold, additional properties of the author such as his affiliation could be leveraged to resolve such name conflicts in some cases.

Two queries are needed to extract this information from the article collection. First we need to establish a b-node for each *distinct* author name:

```
CONSTRUCT
  DISTINCT-NAMES{ all distinct name[var AName] }
FROM
  bib{{ _ {{ # Article, Inproceeding, Techreport, ...
         desc author{{ name{ var AName } }} }} }}
END
```

An author name is found in the XML article collection in name children of author elements under top-level entries representing articles, theses, technical reports, etc. Since there may be elements between entry and author (e.g., authorgroup or affiliation) Xcerpt's **desc** modifier is used to indicate that authors at any depth are to be included. In the construct part of the rule, a temporary store for the names is created that contains one child for each **distinct** binding of the variable AName.

In a second rule, we query the result of the first one: For each pair of authors within the same top-level entry (i.e., publication) the (automatically assigned) ID of the corresponding name element as created by the previous rule is queried and used as the local ID of the b-node for that author.

```
CONSTRUCT
  RDF-STORE{
    all ( triple[value(idvar AID1), "dblp:coauthor",
                 value(idvar AID2)] ) }
FROM
  and (
    DISTINCT-NAMES{{
      idvar AID1 @ name[var AName1]
      idvar AID2 @ name[var AName2] }}
    bib{{ /.*/{{
          desc author{{ name{ var AName1 } }}
          desc author{{ name{ var AName2 } }} }} }} )
```

```
END
```

NORMALIZING RDF OR FROM RDF TO RDF. As mentioned above, a multitude of serializations for RDF in XML and structured text exists and frequently new serialization formats are adopted. Therefore, built-in support for one or even a small number of these serialization formats alone is not sufficient. Alternatively or additionally, a versatile Web query language should provide user definable mappings from arbitrary RDF serializations to a uniform or normalized view of RDF used as target for queries. In other words, the various *physical representations* of RDF must be mapped to a *logical view* of RDF.

This section illustrates such mappings as Xcerpt rules for some of the available RDF serialization, viz. RDF/XML [20], and RDF/A [4]. Fortunately, this can be achieved independently of the schema of the RDF data, which means that the examples of this section work just as well for any other vocabulary than FOAF.

In a semi-structured query language two possible *logical views* of RDF are most reasonable: RDF as (relational) triples and as proper graph.[5] For many queries the second view is more favorable, as it allows the leveraging of expressive graph traversal operators such as **descendant** or regular path expressions in queries. However, for simplicity a triple view of RDF as in the above examples is assumed in the following. For a more detailed discussion of choosing an appropriate logical view on RDF see [98].

TRANSFORMING RDF/XML TO TRIPLES. The standard serialization format for RDF is RDF/XML [20], a W3C recommendation since 2004 very close to the original 1999 RDF syntax. Surprisingly, it is very difficult to parse this serialization format as it has a high degree of variability. This originates partially from the design goal that the syntax allows terse statements of large XML graphs without unnecessary repetition or duplication in the syntax leading to a large number of abbreviations and purely syntactical variants, making reading and processing of RDF/XML non trivial. The following example document shows a few fictive statements in RDF/XML about "Marcus T. Cicero" based on the FOAF vocabulary:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF ... >
  <jur:Lawyer rdf:about="people:m_t_cicero"
```

5 Obviously, any structure in between could also be chosen, however, as [197] shows it is far from obvious to choose a good "slicing" of the RDF graph that determines which relations are expressed through direct links and which through value references.

```
          foaf:name="Marcus T. Cicero">
  <foaf:member rdf:resource="pol:Optimates" />
  <foaf:depiction>
    <rdf:Description>
      <foaf:creator
        rdf:resource="people:m_t_cicero" />
    </rdf:Description>
  </foaf:depiction>
</rdf:Description></rdf:RDF>
```

Description elements represent resources occurring as subjects in RDF triples. They contain elements or attributes that define their properties. The object of a statement is attached as attribute value, as element content, or as value of the special rdf:resource attribute. Thus, the above RDF/XML document defines the following RDF triples (in Turtle notation).

```
people:m_t_cicero foaf:name "Marcus T. Cicero".
people:m_t_cicero rdf:type <jur:Lawyer>.
people:m_t_cicero foaf:member pol:Optimates.
people:m_t_cicero foaf:depiction _:bust_17.
_:bust_17 rdf:creator people:m_t_cicero .
```

This example gives only a glimpse at the many variants allowed in RDF/XML. For more details on the variants and a full description on how to use Xcerpt to transform RDF/XML in a triple view of RDF are given in [38].

A brief look at one of the transformation rules suffices to demonstrate the level of versatility needed to integrate such formats:

```
CONSTRUCT
  RDF-STORE{ all triple[var SURI, var PURI, var OURI] }
FROM
  and(
    rdf-subjects {{
      idvar S @ _{{ var PURI (( rdf:resource=var OURI )){{
      }} }} }},
    node-to-triple-value[ idvar Subject, var SubjURI ] )
END
```

The rule uses two helper rules rdf-subjects and node-to-uri to find all subject resources in the RDF/XML document (this requires a recursive traversal of the document, as subject resources may occur at any depth and are only distinguishable from properties and objects through their structural position). The second helper rule is node-to-uri that associates nodes in the RDF/XML document with URIs (needed to resolve relative URIs, assign "URIs" to blank nodes etc.). The above rule selects for each subject node the immediate children of that node, which represent the properties of the

subject node. Finally, the URI of the object is selected from the value of the rdf:resource attribute of the property. Obviously, this rule only covers one of the many cases how triples are represented, it can, e.g., not handle literal objects or nested objects.

TRANSFORMING RDF/A TO TRIPLES. RDF/A [4] is a recent W3C editor's draft proposing a new serialization of RDF that allows to embed RDF statements as attributes in any possible XML markup language, such as XHTML or SVG. An example RDF/A fragment is shown in the following listing.

```
<p about="http://senate.spqr/m_t_cicero">
For many years, <span property="foaf:name">Marcus T.
Cicero</span> is a recognized name, both as a <span
property="rdf:type" href="jur:Lawyer">lawyer</span> and as
a senator. He is a member of the conservative <a
property="foaf:member" href="pol:Optimates">Optimates
party</a>. He has also created <span href="[_:bust_17]"
rev="foaf:depiction" rel="foaf:creator">Bust 17</span>
depicting himself.</p>
```

This RDF/A fragment represents the same triples as the above RDF/XML document: Subjects of statements are indicated by the about attribute, predicates by one of the attributes property (if the object of the statement is a literal), rel (if the object of the statement is a URI) and rev (if the statement is to be read in the reverse direction). In case the objects of statements are literals, in RDF/A they are either included in the element with the subject and predicate attributes or in a content attribute, which takes precedence. If the object is an URI, it is included in an href attribute.

The different ways RDF triples may be embedded in XHTML (or any other XML markup language) can be covered in the disjuncts of a single Xcerpt rule:

```
CONSTRUCT
RDF-STORE{ all triple[ var S, var P, var O] }
FROM
or (
 desc _((about=var S, property=var P, without content=_))
      {{var Object}}
 desc _((about=var S, rel=var P, href=var O)){{ }}
 desc _((about=var O, rev=var P, href=var S)){{ }},
 desc _((about=var S, property=var P, content=var O)){{}})
END
```

Not all possible embeddings of triples in the RDF/A syntax are covered by this rule: Is the about attribute absent for an element with a property

attribute, the subject of the corresponding statement is resolved by *subject resolution*, cf. Section 3.3.4.

There are also other, non-W3C RDF serialization formats that are more regular and become very similar to the logical triple view of RDF discussed in this section.

FROM TRIPLES TO GRAPHS. Given the triple view, one can formulate easily expressive queries against the RDF data. However, whenever the queries involve path traversals, in particular arbitrary length path traversals (e.g., to traverse the transitive closure of a relation) complex and often recursive rules are needed.

However, if RDF is considered as a graph, where similar as in RDF/XML subjects contain properties which in itself contain links to objects of statements, then such queries can be expressed with **descendant** or regular path expressions as available in most XML and semi-structured query languages.

In the spirit of versatility, Xcerpt provides access to both logical views. The following rule transforms the triple into a graph view.

```
CONSTRUCT
  RDF-GRAPH-STORE {
    all var Subject @ var Subject {
      all optional var Predicate { ^var Object },
      all optional var Predicate { var Literal } } }
FROM
  RDF-TRIPLE-STORE[
    triple[ var Subject, var Predicate,
          optional var Literal → literal{{}},
          optional var Object → /.*/ ] ]
END
```

*From Topic Maps to RDF*

Topic Maps being an ISO standard fitting similar purposes as RDF, it is often desirable to draw information from both of these semantic Web data formats simultaneously. The large amount of research aiming at easing the interoperability amongst both formats, cf. [178], is an indicator for the necessity of a versatile query language like Xcerpt that allows the aggregation of information from both formats.

A possible procedure for integrating Topic Maps with other formats would be the transformation of topics and associations to sets of triples in a similar way as the transformation of the DBLP-like article collection above. There are several kinds of triples that may be extracted from a topic map:

the type of the topic, its name, and named occurrences of the topic. The transformation of topics to RDF-like triples—and therefore the integration of information from Topic Maps and RDF—is obtained quite naturally using a format versatile query language and is therefore omitted for space reasons.

### 3.3.3 SCHEMA VERSATILITY

Schema versatility builds upon format versatility in the sense that the integration of information from different resources in many cases requires that the employed query language is both format versatile and schema versatile. In fact, it is unlikely that pieces of information gathered from sources in different formats make use of the same schema. As an example reconsider the integration of information from the DBLP-like article collection and FOAF descriptions. While both sources of information have been brought into a uniform triple notation thanks to format versatility, the schemata of both sources remain unassociated. This is where schema mappings specified, e.g., in the W3C's RDFS vocabulary definition language or in the Web Ontology Language (OWL) come into play.

Both languages provide some means to establish a mapping between classes and properties in different schemata, though the mapping concepts of RDFS are very limited. A schema mapping might, e.g., state that dblp:coAuthorOf is a subproperty of foaf:knows, i.e., that all resources that stand in dblp:coAuthorOf relation also stand in foaf:knows relation.

If the query language supports reasoning with RDFS and/or OWL, it suffices to include such schema mappings into the considered data. The query engine then infers the appropriate tuples. Xcerpt provides such reasoning support for RDFS in form of a rule library that also illustrates how user defined schema mappings (e.g., going beyond the mapping constructs of either RDFS or OWL) can be supported in a query language. Xcerpt's RDFS rule library is described in more detail in [38, 98]. Here, it suffices to give an impression of the kind of rules needed to realize RDFS reasoning:

```
CONSTRUCT
  RDFS-STORE{
    optional all triple[var Subject, var SuperPr, var Object]
    optional all var BasicTriple }
FROM
  or (
    RDFS-STORE{{
      triple[var SubPr, "rdfs:subPropertyOf", var SuperPr]
      triple[var Subject, var SubPr, var Object] }}
    RDFS-STORE{{ var BasicTriple }} )
```

```
11 END
```

The rule queries the RDF-STORE for all triples with predicate rdfs:subPropertyOf. Such triples connect a sub-property `SubPr` to a super-property `SuperPr`. In the inferred RDFS-STORE a new triple is inserted for each basic triple with the sub-property as predicate. Additionally, all the basic triples are included as well.

Beyond simple equivalences or specialization relations, schema mappings may contain more elaborate information, e.g., that a property of an object is a primary key, i.e., its values uniquely identify that object. In OWL this is specified by typing the property as owl:InverseFunctionalProperty. This information can be used to recognize that two objects, even if they are identified differently (most commonly at least one of them is a blank node) are indeed the same. In the example case, ISBNs or DOIs of books and articles qualify for inverse functional properties. This allows to infer equivalence of individual books even if the schema mapping contains only equivalences on properties and classes.

Schema versatility often goes hand in hand with the two other forms of versatility: Often different schemata originate from different formats used for the data; often different schemata use different representations for the same data. The link between schema and representational versatility is further investigated in the following section.

### 3.3.4 REPRESENTATIONAL VERSATILITY

Even within the same schema, the represenation of information may vary to a great extent, and the semi-structured nature of data on the Web requires that an adequate query language can handle heterogeneous and incomplete data and complex nested structures. RDF/A is an example for an XML schema that allows a great degree of representational diversity. Especially the concept of subject resolution, which has been mentioned in Section 3.3.2, requires that a Web query language that is supposed to handle RDF/A is representationally versatile. Subject resolution in RDF/A means that in the absence of an about attribute, the subject of a statement is searched for as the value of the nearest available about attribute of enclosing elements.

In order to extract also triples of this kind, the rule from Section 3.3.2 must be adjusted to use Xcerpt's regular path expressions. The second disjunct of the above rule would read as follows.

```
1 desc _ (( about=var Subject )){{
  desc(!(_[about=_]))*
3   (( rel=var Predicate, href=var Object )){{ }}
```

```
}}
```

The FOAF vocabulary specification provides many different ways to specify the name of a person, such as foaf:firstName, foaf:nick, foaf:givenname, foaf:family_name, foaf:name, and foaf:surname. The automatic creation of an address book from a set of FOAF descriptions requires that all or some of these possibilities are taken into account. Undoubtedly, a representationally versatile query language must provide a construct that allows certain parts of semi-structured data to be *optional*, in the sense that they are to be retrieved if present, but that the query need not fail if they are absent. The following Xcerpt rule transforms a set of FOAF descriptions into an address book making intensive use of the **optional** construct.

```
CONSTRUCT
  addressbook[
    all address[
      mbox[ var Mbox ],
      firstname[ optional var FirstName, optional var GivenName ],
      familyname[ optional var FamilyName, optional var Surname ],
      optional name[ var Name ] ] ]
FROM
  foaf:Person{{
    foaf:mbox{ var Mbox },
    optional foaf:firstName{ var FirstName },
    optional foaf:family_name{ var FamilyName },
    optional foaf:surname{ var Surname },
    optional foaf:name{ var Name },
    optional foaf:givenname{ var GivenName } }}
END
```

In construct terms, **optional** marks the enclosed subterms as optional, i.e., they are only included in the result, if there are bindings for the free variables in the scope of the **optional**. Therefore, the element name in the above rule is only included, if the query part of the rule succeeded to match the Name variable.

Note that the rule is written based on the assumption that there is no major semantic difference neither between surnames and family names, nor between first names and given names. Furthermore, it is assumed that a single FOAF description does not contain both a family name and a surname or a first name and a given name. If this is not the case, a precedence could be given to, e.g., firstName and familyName and realized with Xcerpt's conditional construction.

Concluding, the above examples illustrate how Xcerpt is a first step towards realizing the vision of versatile query languages outlined in Chapter 2 and makes access to heterogeneous data sources almost as easy as

access to homogeneous sources. This versatile nature of Xcerpt is further strengthened by a number of features discussed in the following sections.

## 3.4 ADDING IDENTITY: FROM HERAKLIT TO CODD

Managing structured or semi-structured data involves the determination of what defines the identity of a data item (be it a node in a tree, graph, or network, an object, a relational tuple, a term, or an XML element). Identity of concepts in data management, most notably for joining, grouping and aggregation, as well as for the representation of cyclic structures.

"What constitutes the identity of a data item or entity?" is a question that has been answered, both in philosophy and in mathematics and computer science, essentially in two ways: based on the extension (or structure and value) of the entity or separate from it (and then represented through a surrogate).

EXTENSIONAL IDENTITY. Extensional identity defines identity based on the extension (or structure and value) of an entity. Variants of extensional identity are Leibniz's law[6] of the *identity of indiscernibles*, i.e., the principle that if two entities have the same properties and thus are indiscernible they must be one and the same. Another example of this view of identity is the *axiom of extensionality* in Zermelo-Fraenkel or Von Neumann-Bernays-Gödel set theory stating that a set is uniquely defined by its members.

Extensional identity has a number of desirable properties, most notably the compositional nature of identity, i.e., the identity of an entity is defined based on the identity of its components. However, it is insufficient to reason about identity of entities in the face of changes, as pointed out by Heracleitus around 500 B.C.:

> *ποταμοῖσι τοῖσιν αὐτοῖσιν ἐμβαίνουσιν ἕτερα καὶ ἕτερα*
> *ὕδατα ἐπιρρεῖ·*
>
> You cannot step twice into the same river; for fresh waters are flowing in upon you. (HERACLEITUS, *Fragment 12*)

He postulates that the composition or extension of an object defines its identity and that the composition of any object changes in time. Thus, nothing retains its identity for any time at all, there are no persistent objects.

**6** So named and extensively studied by Willard V. Quine.

This problem has been addressed both in philosophy and in mathematics and computer science by separating the extension of an object from its identity.

SURROGATE IDENTITY. Surrogate identity defines the identity of an entity independent from its value as an external surrogate. In computer science surrogate identity is more often referred to as *object identity*. The use of identity separate from value has three implications (cf. [14] and [140])

- In a model with surrogate identity, naturally two notions of object *equivalence* exist: two entities can be identical (they are the same entity) or they can be equal (they have the same value).
- If identity is separate from value, combining the same two parts multiple times may lead to entities of different identity. That is, it is possible that two distinct entities *share* the same (meaning identical, not just same value) properties or sub-entities.[7]
- *Updates* or changes on the value of an entity are possible without changing its identity, thus allowing the tracking of changes over time.

In [14] value, structure, and location independence are identified as essential attributes of surrogate identity in data management. An identifier or identity surrogate is value and structure independent if the identity is not affected by changes of the value or internal structure of an entity. It is location independent if the identity is not affected by movement of objects among physical locations or address spaces.

Object identity in object-oriented data bases following the ODMG data model fulfill all three requirements. Identity management through primary keys as in relational databases violates value independence (leading to Codd's extension to the relation model [74] with separate surrogates for identity). Since object-oriented programming languages are usually not concerned with persistent data, their object identifiers often violate the location independence leading to anomalies if objects are moved (e.g., in Java's RMI approach).

Surrogate or object identity poses, among others, two challenges for query and programming languages based on a data model supporting this form of identity: First, where for extensional identity a single form of equality (viz. the value and structure of an entity) suffices, object identity induces at least two, often three flavors of *equality* (and thus three different *joins*): Two entities may be equal wrt. identity (i.e., their identity surrogates

7 In other words, composition is no longer an identity preserving operation.

are equivalent) or value. If entities are complex, i.e., can be composed from other objects, one can further distinguish between "shallow" and "deep" value equality: Two entities are "shallow" equivalent if their value is equal and their components are the same objects (i.e., equal w.r.t. identity) and "deep" equivalent if their value is equal and the values of their components are equal. Evidently, "shallow" value-based equality can be defined on top of identity-based and "deep" value-based equality.

The same distinction also occurs when *constructing* new entities based on entities selected in a query: A selected entity may be linked as a component of a constructed entity (object sharing) or a "deep" or "shallow" copy may be used as component.

Summarizing, surrogate or object identity is the richer notation than extensional identity addressing in particular object sharing and updates, but conversely also requires a slightly more complex set of operators in query language and processor.

Following a short outlook at related work on object identity in data management, the advantages and challenges for introducing surrogate identity in Xcerpt 2.0 are investigated.

### 3.4.1 OBJECT IDENTITY IN DATA MANAGEMENT

The need for surrogate identity in contrast to extensional identity as in early proposals of the relational data model has been argued for by Codd in [74], as early as 1979. He acknowledges the need for unique and permanent identifiers for database entities and argues that user-defined, user-controlled primary keys as in the original relational model are not sufficient. Rather permanent *surrogates* are suggested to avoid anomalies resulting from user-defined primary keys with external semantics that is subject to change.

In [140] an extensive review of the implications of object identity in data management is presented. The need for object identity arises if it is desired to "distinguish objects from one another regardless of their content, location, or addressability" [140]. This desire might stem from the need for dynamic objects, i.e., objects whose properties change over time without loosing their identity, or versioning as well as from object sharing.

[140] argues that identity should neither be based on address (-ability) as in imperative programming languages (variables) nor on data values (in the form of identifier keys) as in relational databases, but rather a separate concept maintained and guaranteed by the database management system.

Following [140], programming and query languages can be classified in two dimensions by their support for object identity: the first dimension represents to what degree the identity is managed by the system vs. the

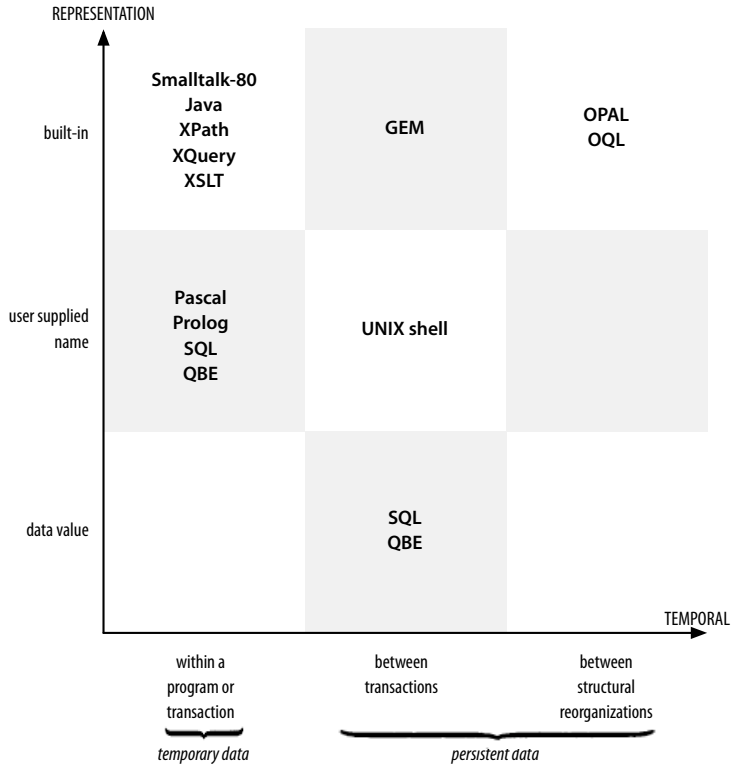



Figure 4. Identity in Programming and Query Languages

user, the second dimension represents to what degree identity is preserved over time and changes, see Figure 4.

Problems of user defined identity keys as used in relational databases lie in the fact that they cannot be allowed to change, although they are user-defined descriptive data. This is especially a problem if the identifier carries some external semantics, such as social security numbers, ISBNs, etc. The second problem is that identifiers can not provide identity for some subsets of attributes.

The value of object identities (OIDs) as query language primitives is investigated in [1]. It is shown that OIDs are useful for

- object sharing and cycles in data,
- set operations,
- expressing any computable database query.

The data model proposed in [1] generalizes the relational data model, most complex-object data models, and the logical data model [145]. At the core of this data model stands a mapping from OIDs to so-called *o*-values, i.e., either constants or complex values containing constants or further OIDs. Repeated applications of the OID-mapping yield *pure values* that are regular infinite trees. Thus trees with OIDs can be considered finite representations of infinite structures.

The OID-mapping function is partial, i.e., there may be OIDs with no mapping for representing incomplete information.

It is shown that "a primitive for OID invention must be in the language … if unbounded structures are to be constructed" [1]. Unbounded structures include arbitrary sets, bags, and graph structures.

Lorel [3] represents a semi-structured query language that supports both extensional and object identity. Objects may be shared, but not all "data items" (e.g., paths and sets) are objects, and thus not all have identity. In Lorel construction defaults to object sharing and grouping defaults to duplicate elimination based on OIDs.

### 3.4.2 OBJECT IDENTITY IN XCERPT 2.0

In the classification scheme established in the previous section, previous versions of Xcerpt supports extensional identity exclusively. Before presenting arguments for surrogate identity, a closer look at the data model of Xcerpt 1.0 as defined in [188] is needed.

```
article [
 sect [
  para [ "Wie froh ..." ]
  title [ "First" ]
  intro @ sect [
   title [ "First" ]
   sect [
    title [ "First" ]
    ^intro
   ]
  ]
 ]
]
```

```
article [
 sect [
  para [ "Wie froh ..." ]
  title [ "First" ]
  intro @ sect [
   title [ "First" ]
   sect [
    title [ "First" ]
    sect [
     title [ "First" ]
     sect [
      title [ "First" ]
      ^intro
     ] ] ] ] ] ]
```

Figure 5. Cyclic Xcerpt Data Terms

*Xcerpt 1.0: Regular Infinite Trees*

In contrast to claims in prior descriptions of Xcerpt 1.0, the *data model* of Xcerpt is indeed not much so based on graphs, as on regular infinite trees. This is in part due to the lack of object sharing at the level of the data model (rather than only in the serialization of data) and thus very much related to the topic of this article. After [80] a (possibly infinite) tree $t$ is called *regular*, if under structural equality the set of all its (possibly infinite) subtrees is finite. Acyclic Xcerpt 1.0 data terms can be seen as finite regular trees, cyclic Xcerpt 1.0 data terms as infinite regular trees, since their number of subtrees is finite under structural equality. Consider, e.g., the two data terms in Figure 5.

Both data terms have the same five subtrees, viz. the subtree rooted at article[1], sect[2], para[3], title[4], and sect[5]. Evidently, the number of explicit representations of the cycle in the data term does not affect the number of subtrees.

This result is unsurprising in the light of [1], where it is shown that a graph-shaped object-oriented data model (with object sharing) can be reduced to a regular infinite tree, if one ignores object identity replacing (recursively) each object reference with the value of the object. The informed reader will notice that the latter is the conceptual modus operandi of Xcerpt 1.0.

The cause of this limitation is a desirable property of Xcerpt 1.0, viz. that parent-child and ID/IDREF-links are indistinguishable in the data

model and in queries. This means in other worlds, that Xcerpt does not distinguish between object copy and object reference or sharing.

In a pure *tree data model* this is indeed no limitation at all, since in a tree data model the position of an object (i.e., the position among its siblings and (recursively defined) the identity of its parent node) is sufficient for a unique identification of that object. However, *in graphs* this does not suffice due to object sharing, i.e., the occurrence of the identical object at different positions. Positional identity would, e.g., in the data term `a{ x@b{}, ^x}` result in two `a` children of the `b` with different identity. Considering the example `x@a{^x, ^x}` one sees how this "positional" identity leads to infinitely many objects, if the term contains cyclic references, and thus to infinite (but regular) trees. This is also the root cause why there is no Xcerpt query that can distinguish between the data term `a{ x@b{}, ^x}` and the data term `a{ b{}, b{} }`. Neither is there a query that can distinguish the two data terms in Figure 5.

Indeed, positional identity is already used in Xcerpt 1.0 to some extent, viz. in the index injectivity property of the simulation relation.

To sum up, currently Xcerpt 1.0 uses a data model based on regular infinite trees but for serialization (and in-memory representation) these regular trees are represented as finite and, in general, cyclic graphs. The following section argues that there are at least three reasons to uses full object identity, as Xcerpt 2.0 does, and, as an immediate consequence, to use a proper graph data model.

*Object Identity: Updates, Sharing,*

Historically, there are two main incentives for object identity vs. extensional identity in data management systems, viz. object sharing and object updates. Both apply also for Xcerpt. Additionally, object identity is needed in Xcerpt to properly support an important class of queries, viz. occurrence queries.

OBJECT UPDATES. Under extensional identity object updates and transformations are indistinguishable. In particular, it is not possible to *track* changes to objects over time. Object identity, on the other hand, allows the *tracking* of object updates. E.g., event queries of the form "if a certain object is changed twice within 10 minutes" require some form of object identity. As argued above, it is possible to simulate object identity through extensional identity, but this leads to a violation of the value independence and places the burden of managing the identity on the user.

OBJECT SHARING. If object identity is provided in a data management system, object sharing is almost always desirable to make views, rules, or similar "procedural abstraction" mechanisms transparent. Without object sharing, such procedural abstractions have a side-effect, namely that all objects in their result get a new identity, even if they are extracted from queried data. This makes identity joins over rule boarders infeasible (see XQuery's violation of referential transparency by combination of element construction and **let**, cf. Section 2.4.1).

**Occurrence Queries:** Possibly the strongest argument for object identity is, however, the need for occurrence queries. Occurrence queries ask for occurrences of certain data items in the data base. Examples of occurrence queries are, e.g., *"How many title elements occur in the first section?"*, *"List all occurrences of title in the first section!"*, or *"For each supplier count the average number of items per order (where an order may contain the same item several times)!"*.

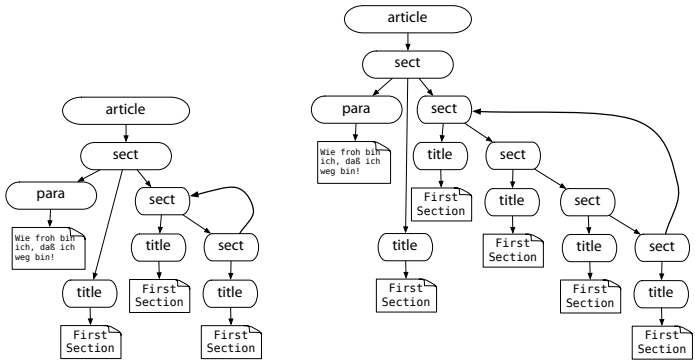
In Xcerpt occurrence queries are only supported as "distinct" occurrence queries, i.e., in the form *"How many distinct title elements occur in the first section?"*. It is not possible to count the actual occurrences, if some of them share the same structure, as they are considered identical in this case. One might think that a simple solution here could be to use bags to gather these occurrences (instead of sets as in the current semantics of Xcerpt). This, however, suffices not as a solution: Consider, e.g., the following query term against the first data term of Figure 5:

```
article [[
  position 1 sect {{
    var X → desc title {{ }}
  }}
```

The query asks *"Find all occurrences of title elements at any depth in the first section."*

How many bindings for X should this query return on the given data? If one assumes duplicate elimination based on extensional identity (structure and value of terms), a single binding for X is returned. If no duplicate elimination takes place, however, i.e., in the case of an occurrence query where simply all the occurrences are of interest, there are *infinite occurrences* under the section. This becomes more obvious if one considers the same data with a longer cycle as in the second data term of Figure 5. To better illustrate the point, Figure 6 shows a visual representation of the data.

As argued above, in Xcerpt 1.0 both data terms represent the same infinite regular tree and thus the answer to this query must be the same. No bound for the number of occurrences of title in the first section can be given. Indeed, this is the expected answer, if one assumes a data model of




(a) Short Cycle (b) Long Cycle

Figure 6. Structure-equivalent Data Terms with Different Cycle Length

regular infinite trees as Xcerpt does (see above).

If one assumes duplicate elimination based on object identity, the query returns different answers on the two data terms, as they are no longer identical (they represent different graphs), viz. three bindings for the first data term and five for the second.

The following two tables summarize the differences between the result for the query term on the two data terms with different forms of duplicate elimination:

| duplicate elimination | **X** |
|---|---|
| value-based | `title[ "First"]` |
| identity-based | $t^4, t^6, t^8$ |
| none | $t^4, t^6, t^8, t^{10}, t^{12}, t^{14}, t^{16}, \ldots$ |

| duplicate elimination | **X** |
|---|---|
| value-based | `title[ "First"]` |
| identity-based | $t^4, t^6, t^8, t^{10}, t^{12}$ |
| none | $t^4, t^6, t^8, t^{10}, t^{12}, t^{14}, t^{16}, \ldots$ |

To conclude, unrestricted occurrence queries are not expressible in Xcerpt 1.0 (only distinct occurrence queries are). But unrestricted occurrence queries are an essential class of queries. Their introduction on a regular infinite tree data model is however at least problematic, at worst

impossible. On a graph data model with object identity they are easy to handle, but an obvious consequence is the loss of Xcerpt's property to handle reference and child links indistinguishably.

*Supporting Identity in Xcerpt 2.0: Language Constructs*

Xcerpt 2.0, as defined in Section 3.2 and in [101] uses node identity and a proper graph model with an equivalence based on node identity (in addition to label and deep equality). To support this data model Xcerpt is extended by the following constructs:

**(1)** **Identity variables** are (temporary) representatives for the identity surrogate of a node. In query terms, they may occur instead of term identifiers, e.g., `desc idvar ID @ a {{}}` and on term level (like normal variables) in which case they match with the identity surrogate of any node that can occur at that position. To separate them from other variables in query terms, this is not true for construct terms. Therefore a different keyword, e.g., **`idvar`**, is used to emphasize the difference.

**(2)** Queries may contain **identity joins** expressed simply by repeated occurrence of the same identity variable. This means that queries can define the length of the cycle, but arbitrary cycle length queries can still be expressed using **`descendant`** or qualified descendant. Cyclic queries using "pseudo identifiers" are treated in the same way but do not allow the propagation of identity to the head of a rule. Xcerpt 2.0 cyclic queries are not only easier to evaluate, but also have a clear strict and immediate meaning.

**(3)** Joins on non-identity variables remain pure value joins.

**(4)** In construct terms, **object sharing** is achieved by using identity variables instead of normal variables, indicating that a link to the object represented by the surrogate is to be constructed. A normal variable still indicates just a deep copy of the value.

**(5)** When creating unbounded structures, i.e., in the case of grouping, *surrogate invention* is needed: E.g., if the result should contain an `a` with itself as child for each binding of some variable the link between the two must be established using surrogate invention. The syntax in Xcerpt 2.0 is as follows:

```
CONSTRUCT
result [
  all new_id(invention_label) @ a [
    ^last_id(invention_label)
  ] group-by var X
]
```

```
FROM
var X → desc title {{ }}
END
```

More complex object sharing structures can be expressed by rule chaining.

The above extension leaves the issue of serialization of identity surrogates to the implementation. For XML, we expect the use of ID/IDREF attributes, but the shape of the ids is left to the implementation.

## 3.5 MODULES: FROM SEPARATION TO ENCAPSULATION

Modules are software units that group together parts of different programs and knowledge (or data structures) and that usually serve a specific purpose. For any practically applied programming language modules are an essential feature, as they make large scale projects tractable by *humans*. Modules not only facilitate the integration of existing applications and software into complex projects, they also offer a flexible solution to application development where part of the application logic is subject to change.

The work presented in this chapter first advocates the need and advantages of modularizing rule-based languages. The rule-based paradigm offers elegant and flexible means of application programming at a high-level of abstraction. During the last couple of years, a considerable number of initiatives have focused on using rules on the Web (e.g. RuleML, R2ML, XSLT, RIF Core etc.).

However, unless these rule languages are conceived with proper support for encoding knowledge using modular designs, their contributions to the Web are arguably doomed to exist in isolation, hence with no easy mechanism for integration of valuable information. Many of the existing rule languages on the Web do not provide support for such modular designs. The rationale behind this is the focus on the initially more crucial concerns relating to the development of the languages.

The main difference between the module system presented here and previous work on module systems is, that our approach focuses on genericity. That is, the approach is applicable to many rule languages with only small adaptation to syntactic and semantic properties of the language to be supported with constructs for modularization. The concept is based on rewriting modular programs into semantically equivalent non-modular programs, hence not forcing the evaluation of the hosting language to be adapted to some operational module semantics. This approach not only enables reuseability for current languages, it is arguably also applicable to forthcoming rule languages. The presented module system hence ar-

guably enables reuseability in two aspects—it not only supports users of languages to design programs in a modular fashion, it also encourages tool and language architects to augment their rule languages by reusing the abstract module system. Last but not least: the module system is *easy* in several aspects. First, it is very easy for language developers to apply due to the employment of reduction semantics of a given modularized rule language to its un-modular predecessor. Second, the reduction semantics is kept simple. There is just one operator, yet it is sufficient to model a quite rich modular landscape including visibility, scoped module use, parametric modules etc., though we disallow recursive modules, for reasons of simplicity. Third, the implementation of the abstract module system can be achieved using existing language composition tools, for example, Reuseware.[8] A concrete modularized language is achieved by mere instantiation of the abstract implementation, making the implementation of the abstract module system fast and easy.

The abstract module system is then applied to extend Xcerpt with modules. Modules allow a *separation of concern* not just on the basis of single rules but on the basis of larger conceptual units of a query program. For example, one part of a Web application is often concerned with extracting data from a set of sources, such as a set of Web pages. At the next step, the data might have to be syndicated into a common view and format. From this syndicated data, some new implicit data could possibly be derived. Finally, the resulting data set should be displayed in an appropriate human-readable form, for example, by being displayed in a well-structured Web page (see Section 3.5.4 for an example). These different steps taken by the application have to do with different concerns of the overall realization, such as data extraction, data management and data display. Furthermore, each of the concerns deals with different schemata, but the knowledge of the schemata can be hidden and encapsulated within each concern – within each module. In contrast, exposing all these concerns in one monolithic query program not only becomes very hard to understand, but is also impossible to manage as a change in some part may affect any other part.

For Xcerpt, we propose a module system that (a) demonstrates how Web query languages can profit from modules by partitioning the query program as well as its execution; (b) provides an easy, yet powerful module extension for Xcerpt that shows how well-suited rule-based languages are for component-based reuse; (c) is based on a single new concept, viz. "stores"; and (d) uses a reduction semantics exploiting the power of a

8 http://reuseware.org

language with views. This semantics enables the reuse of the existing query engine making the design of the module system easier and its deployment less time consuming.

### 3.5.1 MODULE EXTENSION BY EXAMPLE

Rules in a particular language are usually specified in a *rule set*, a finite set of (possibly) related rules. Recall from above, that we focus our work on so-called deductive rules. A *deductive rule* consists of a *head* and a *body* and has the following form:

$$head :- body$$

where $body$ and $head$ are finite sequences of atomic formulas of the rule language. The formulas of each set are composed using (language specific) logical connectives. The implication operator :– binds the two rule parts together. Rules are read "if the body holds, then the head also holds". Usually, rule parts share variables: the variable substitutions obtained by evaluating the *body* are used to construct new data in the *head*.

In the following we extend a concrete rule language with modules. By following our proposed approach, which is formalized in Section 3.5.3, one can afford to abstract away from a particular data model or specific capabilities supported by a rule language and - most important - not to change the semantics of the language.

*Datalog as an Example Rule Language*

In the introductory part we mentioned Datalog as an example rule language based on deductive rules. Datalog is a well-known database query language often used for demonstrating different kinds of research results (e.g. query optimization). Datalog-like languages have been successfully employed, e.g. for Web data extraction in Lixto[9]. The strong similarities between Datalog and RIF Core suggest that the ideas followed for modularizing Datalog could be also applied to RIF Core.

In Datalog, the $head$ is usually considered to be a singleton set and the $body$ a finite set of Datalog *atoms* connected via conjunction ($\wedge$). A Datalog atom is an $n$-ary predicate of the form $p(a_1, \ldots, a_n)$, where $a_i (1 \leq i \leq n)$ are constant symbols or variables. As such, a Datalog rule may take the following form:

**9** Data extraction with Lixto, http://www.lixto.com/li/liview/action/display/frmLiID/12/

```
gold-customer(Cust) :- rentals(Cust,Nr,2006), Nr > 10.
```

to be understood as defining that *Cust* is a gold customer if the number *Nr* of car rentals *Cust* made in 2006 is greater than 10.

Rules are associated a semantics, which is specific to each rule language and cannot be described in general terms. For the case of Datalog, semantics is given by definition of a least Herbrand model of a rule-set. We don't go into details regarding the semantics, since our work on modularizing Datalog preserves the language's semantics.

Datalog, and more general deductive rules, infer new knowledge (called intensional knowledge) from existing, explicitly given knowledge. As already recognized [141], there is a need to 'limit' the amount of data used in performing inference on the Web – a big and open source of knowledge. Thus, the notion of *scoped inference* has emerged. The idea is to perform inference within a scope of explicitly given knowledge sources. One elegant solution for implementing scoped inference is to use *modules* for separating the knowledge. In such a case, the inference is performed within a module. Since inference is essential on the Web and, thus, modules for rule languages such as Datalog, let's see how we could modularize Datalog!

### *Module Extension for Datalog*

This section gives a light introduction to modularizing a rule language such as Datalog that should ease the understanding of the formal operators proposed in Section 3.5.2. We consider as framework for our examples the EU-Rent[10] case study, a specification of business requirements for a fictitious car rental company. Initially developed by Model Systems Ltd., EU-Rent is promoted by the business rules community as a scenario for demonstrating capabilities of business rules products.

The concrete scenario we use for showing advantages of introducing modularization in rule languages is similar to the use case for rule interchange 'Managing Inter-Organizational Business Policies and Practices', published by the W3C Rule Interchange Format Working Group in the 2nd W3C Working Draft of 'RIF Use Cases and Requirements'. The car rental company EU-Rent operates in different EU countries. Thus, the company needs to comply with existing EU regulations and each of its branches needs also to comply with the regulations of the country it operates in.

The EU-Rent company heavily uses rule-based technologies for conducting its business. This was a straightforward choice of technology from

10 EU-Rent case study, http://www.businessrulesgroup.org/egsbrg.shtml

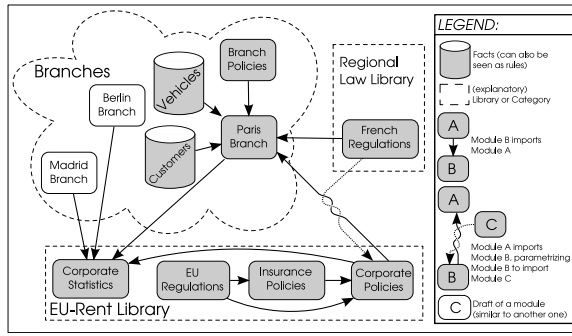



Figure 7. EU-Rent Use Case: Module Structure

the IT landscape, since rule languages are more than suitable for implementing company's policies. Moreover, EU regulations are also given as (business) rules.

Different sets of rules come into play for most of the company's rental services, such as advance reservations for car rentals or maintenance of cars at an EU-Rent branch. A set of rules implement, as touched on above, the company's policies (e.g. that *the lowest price offered for a rental must be honored*). These rules are used by each of the EU-Rent branches. Another set of rules implements the policies used at a particular EU-Rent branch, as they are free to adapt their business to the requirements of the market they address (of course, as long as they remain in conformance with the EU-Rent company-level rules). As is the case for EU regulations, EU-Rent branches might need to comply with existing national regulations—an extra set of rules to be considered.

We have illustrated so far a typical scenario for data integration. The sets of rules our EU-Rent company needs to integrate for its services may be stored locally at each branch or in a distributed manner (e.g. company level rules are stored only at EU-Rent Head Quarter and EU and national rule stores exist on different servers for the corresponding regulations). Rules might change at the company level and regulations might also change both at EU and at national level. So as to avoid the propagation of such changes every time they occur, a distributed and modularized approach to the EU-Rent implementation would be a suitable solution.

An architectural overview of the scenario described so far is given above. The overview sketches a possible modularization of rules employed by the EU-Rent company. Modules are represented here as boxes. An example EU-Rent branch, the Paris branch, subdivides its vehicle and customer data as well as its policies into different modules, hence separating concerns for

gaining clarity and ease maintenance. The module ParisBranch *imports* the defined modules and, thus, uses their rules together with those defined in ParisBranch. Such module dependencies (i.e., *import*s) are indicated by arrows between boxes.

Modules can be imported and their rules can be defined as *private* or *public*. Private rules are not visible, i.e. the knowledge inferred by such rules can not be accessed directly in modules importing them. A statement `private import M2` in a module M1 makes all rules of M2 private within M1 and thus invisible to the modules importing M1. Rules defined as public can be used directly in modules importing them. By importing a module M as public makes its (public) rules visible in all modules importing M.

In the following we specify an excerpt of the module ParisBranch in Datalog:

```
import private Vehicles
import private Customers
import private BranchPolicies
import private CorporatePolicies ( regional-law =
    "FrenchRegulations" )
public vehicle(X) :- voiture(X).
public vehicle(X) :- bicyclette(X).
```

In the following we turn our attention toa CorporateStatistics, another module depicted in our architecture. This module imports CorporatePolicies, but also all branch modules (such as ParisBranch, MadridBranch, and BerlinBranch), a task doomed to produce vast naming clashes of symbols defined in the imported modules. To overcome the problem, a *qualified import* is to be used: Imported modules get local names that are further used to disambiguate the symbols. Thus, one can smoothly use knowledge inferred by different, imported rules with same heads.

The following example shows how Datalog can be extended with a qualified import. To have an overview over the status of EU-Rent vehicles, the notion of an old vehicle is defined differently for the different branches. The modules ParisBranch, MunichBranch, and MadrdidBranch get the local names m1, m2, and m3, respectively. The local names are associated with the module using @.

```
import private CorporatePolicies
import ParisBranch @ m1
import MunichBranch @ m2
import MadridBranch @ m3
private old-vehicle(V) :- m1.vehicle(V), manufactured(V,Y), Y <
    1990.
private old-vehicle(V) :- m2.vehicle(V), manufactured(V,Y), Y <
    2000.
```

```
private old-vehicle(V) :- m3.vehicle(V), manufactured(V,Y), Y <
    1995.
```

The simple module-based architecture and the given module examples show a couple of advantages of such a module-based approach. We have already touched on the *separation of concerns* when describing the modules of Paris branch as having each a well-defined purpose. Modules such as `EURegulations` and `FrenchRegulations` can be *reused* by other applications too (not only by new established EU-Rent branches, but also by other companies). It could even be published by government agencies on the Web. A module-based implementation is much more flexible and less error-prone than one without modules; this eases considerably the *extensibility* of the implementation.

### 3.5.2 FRAMEWORK FOR RULE LANGUAGE MODULE SYSTEMS

In Section 3.5.1 we saw an example of how it is possible to modularize rule-sets—in this case for Datalog—and that it is important to ensure separation (or encapsulation) of the different modules. We choose to enforce this separation statically, i.e., at compile time, due to our desire to reuse—rather than extend—existing rule engines that do not have an understanding of modules. An added advantage is a clean and simple semantics based on concepts already familiar to the users of the supported rule language.

We are interested in extending what is done for Datalog in Section 3.5.1 to a general framework for rule languages. The notion of modules is arguably important not only for Datalog, but for any rule language lacking such an important concept.

However, our reduction semantics for module operators poses some requirements to the expressiveness of a rule language: To describe these requirements, we first introduce in the following a few notions and assumptions on rule languages that give us formal means to talk about a rule language in general. Second, we establish the single requirement we ask from a rule language to be amenable to our module framework: the provision of rule dependency (or rule chaining).

In the next section, we then use these notions to describe the single operator needed to formally define our approach to modules for rule languages. We show that, if the rule language supports (database) views, that single operator suffices to obtain a powerful, yet simple to understand and realize module semantics. Even in absence of (database) views, we can obtain the same result (from the perspective of the module system's semantics) by adding two additional operators.

RULE LANGUAGES. For the purpose of this work, we can take a very abstract view of a rule language: A rule language is any language where **(1)** *programs* are finite sequences of rules of the language; **(2)** *rules* consist of one head and one body, where the body and the head are finite sequences of rule parts; **(3)** *rule parts*[11] are, for the purpose of this work, arbitrary. They are not further structured though they may in fact be negative literals or complex structures. Body rule parts can be understood as a kind of condition and head rule parts as a kind of result, action, or other consequence of the rule. We use indices to identify a rule within a program as well as head or body parts within a rule. Note, that we do not pose any limitations to the shape of the body parts or the connectors used (conjunction, disjunction, etc.).

RULE DEPENDENCY. Surprisingly, we care very little about the actual shape of rule parts, let alone their semantics. The only critical requirement needed by our framework is that the rule language has a concept of *rule chaining* or *rule dependency*. That is, one rule may depend on another for proper processing.

**Definition 3.3** (Rule dependency)**.** With a program $P$, a (necessarily finite) relation $\Delta \subset \mathbb{N}^2 \times \mathbb{N}^2$ can be associated such that $(r_1, b, r_2, h) \in \Delta$ iff the condition expressed by the $b$-th body part of rule $r_1$ *is dependent* on results of the $h$-th head part of the $r_2$-th rule in $P$ (such as derived data, actions taken, state changes, or triggered events), i.e., it may be affected by that head part's evaluation.

Controlling rule (or rule part) dependency can take different forms in different rule languages: in Datalog, predicate symbol provide one (easy) means to partition the dependency space; in XSLT, modes can serve a similar purpose; in Xcerpt, the labels of root terms; etc. However, the realisation of the dependency relation is left to the rule language. We assume merely that it can be manipulated arbitrarily, though the module system never introduces cycles in the dependency relation if they do not already exist. Thus the rule language does not *need* to support recursive rules for the reduction semantics to be applicable, however, if present, recursive rules pose no challenge. We do, however, assume that the dependencies between *modules* are non-recursive.

Observe, that these rewritings may in fact affect the constants used in the program. However, constants are manipulated in such a way that, for

11 We refrain from calling rule parts literals, as they may be, e.g., entire formulas or other constructs such as actions that are not always considered logical literals.

each module, there is an isomorphism between the rewritten program and the original program. Assuming a generic [2] rule language, this renaming has no effect on the semantics of the program.

The module extension framework requires the ability to express an arbitrary (acyclic) dependency relation, however poses no restrictions on the shape of $\Delta$ for a module-free program. Indeed, for any module-free rule program $P$ the unrestricted relation with all pairs of head parts and body parts forms a perfectly acceptable $\Delta$ relation on $P$.

We only require the ability to express *acyclic* dependency relations as the discussed module algebra does not allow cycles in the module composition for simplicity's sake.

Most rule languages that allow some form of rule chaining, e.g., datalog, SWRL, SQL, Xcerpt, $R_2G_2$, easily fulfill this requirement. However, it precludes rule languages such as CSS where all rules operate on the same input and no rule chaining is possible. Interestingly, though CSS already provides its own module concept, that module concept provides no information hiding, the central aim of our approach: Rules from all imported modules are merged into one sequence of rules and all applied to the input data, only precedence, not applicability, may be affected by the structuring in modules.

REDUCTION SEMANTICS. Why do we impose the requirement to express arbitrary dependency relations on a rule language to be amenable to our module framework? The reason is that we aim for a reduction semantics where all the additions introduced by the module framework are reduced to expressions of the original language. To achieve this we need a certain expressiveness which is ensured by this requirement.

Consider, for instance, the Module "CorporateStatistics" in Section 3.5.1. Let's focus only on old-vehicle and the three vehicle predicates from the three local branches. Using the semantics defined in the following section, we obtain a program containing also all rules from the included modules plus a dependency relation that enforces that only certain body parts of old-vehicle depend on the vehicle definition from each of the local branches. This dependency relation can be *realized* in datalog, e.g., by properly rewriting the predicate symbols through prefixing predicates from each of the qualified modules with a unique prefix.

### 3.5.3 MODULE SYSTEM ALGEBRA

Remember, that the main aim of this work is to allow a rule program to be divided into *conceptually independent collections* (modules) of rules with

well-defined interfaces between these collections.

For this purpose, we introduce in Section 3.5.3 a formal notion of a collection of rules, called "module", and its public interface, i.e., that subset of rules that constitutes the (public) interface of the module. Building on this definition, we introduce an algebra (consisting in a single operator) for composing modules in Section 3.5.3 together with a reduction semantics, i.e., a means of reducing programs containing such operators to module-free programs.

OPERATORS BY EXAMPLE. Before we turn to the formal definitions, let's again consider the EU-Rent use case from Section 3.5.1, focusing on the three modules "CorporateStatistics", "CorporatePolicies", and "ParisBranch".

We can define all the import parts of these modules using the module algebra introduced in the following. We use $A \times B$ for indicating that a module $B$ is imported into module $A$ and $A$ inherits the public interface of $B$ (cf. `import public`), $A \bowtie B$ to indicate private import (cf. `import private`), and $A \rtimes_S B$ for scoped import (cf. `import ... @`) where $S$ are pointers to all rule parts addressing a specific module. Note, that public and private import can actually be reduced to the scoped import if the language provides views.

Using these operators we can build formal module composition expressions corresponding to the surface syntax from Section 3.5.1 as follows:

$$
\begin{aligned}
\mathsf{CorporatePolicies}' &= (\mathsf{CorporatePolicies} \times \mathsf{InsurancePolicies}) \bowtie \mathsf{EURegulations} \\
\mathsf{CorporatePolicies}'_{\mathit{french}} &= \mathsf{CorporatePolicies}' \times \mathsf{FrenchRegulations} \\
\mathsf{ParisBranch}' &= (((\mathsf{ParisBranch} \bowtie \mathsf{Vehicles}) \bowtie \mathsf{Customers}) \bowtie \mathsf{BranchPolicies}) \\
&\quad \bowtie \mathsf{CorporatePolicies}'_{\mathit{french}} \\
\mathsf{CorporateStatistics}' &= (((\mathsf{CorporateStatistics} \bowtie \mathsf{CorporatePolicies}') \rtimes_{(1,1)} \mathsf{ParisBranch}') \\
&\quad \rtimes_{(2,1)} \mathsf{MunichBranch}') \rtimes_{(3,1)} \mathsf{MadridBranch}' \\
\mathsf{MunichBranch}' &= \ldots
\end{aligned}
$$

Thus, given a set of basic modules, each import statement is translated into a module algebra expression that creates a new module, viz. the semantics of the import statement. Unsurprisingly, parameterized modules lead to multiple "versions" of similar module composition expressions that only differ in instantiations of the parameters.

### *Defining Modules*

We use *module identifiers* as means to refer to modules, e.g., when importing modules. Some means of resolving module identifiers to modules

(stored, e.g., in files, in a database, or on the Web) is assumed, but not further detailed here.

**Definition 3.4** (Module)**.** A module $M$ is a triple $(R_{\text{PRIV}}, R_{\text{PUB}}, \Delta) \subset \mathcal{R} \times \mathcal{R} \times \mathbb{N}^4$ where $\mathcal{R}$ is the set of all finite sequences over the set of permissible rules for a given rule language. We call $R_{\text{PRIV}}$ the private, $R_{\text{PUB}}$ the public rules of $M$, and $\Delta$ the dependency relation for $M$. For the purpose of numbering rules, we consider $R = R_{\text{PRIV}} \diamond R_{\text{PUB}}$[12] the sequence of all rules in $M$.

We call a module's public rules its "interface": When importing this module, only these rules become accessible to the importing module in the sense that rule bodies in the importing module may depend also on the public rules of the imported module but not on its private rules. Though the module composition discussed in this section does not rely on any further information about a module, a module should be accompanied by additional information for its *users*: documentation about the purpose of the module, what to expect from the modules interface, what other modules it relies on, etc.

Figure 8 shows an exemplary configuration of a program (which is just a module where $R_{\text{PUB}}$ is empty) together with two modules A and B. Where the program consists of a single sequence of (private) rules, the rules of each of the modules are partitioned into private and public rules. The allowed dependency relation $\Delta$ is represented in the following way: All body parts in each of the areas  ,  , and  are depending on all head parts in the same area and no other head parts. No access or import of modules takes place, thus no inter-module dependencies exists in $\Delta$ between rule parts from one of the modules with each other or with the (main) program.

Notice, that for the dependency within a module the partitioning in private and public plays *no role whatsoever*. Body parts in private rules may access head parts from public rules and vice versa.

### *Module Composition*

Module composition operators allow the (principled, i.e., via their public interface) definition of inter-module dependencies. Our module algebra (in contrast to previous approaches) needs only a single fundamental module composition operator. Further operators can then be constructed

12 $\diamond$ denotes *sequence concatenation*, i.e., $s_1 \diamond s_2$ returns the sequence starting with the elements of $s_1$ followed by the elements of $s_2$, preserving the order of each sequence.

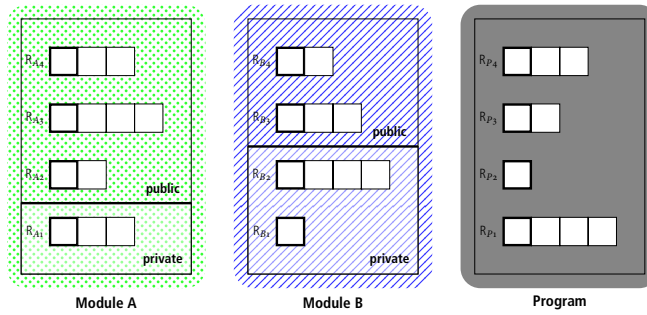



Figure 8. Program and two defined modules without imports

from a combination of that fundamental operator and (standard database) views. However, we also discuss an immediate definition of these operators for languages where the view based approach is not applicable or desirable.

SCOPED IMPORT. The fundamental module composition operator is called the *scoped import operator*. Its name stems from the fact that it allows to specify not only which module to import but also which of the rules of the importing module or program may be affected by that import and thus the scope of a module's import.

Informally, the scoped import of $B$ in $A$ uses two modules $A$ and $B$ and a set $S$ of body parts from $A$ that form the scope of the module import. It combines the rules from $B$ with the rules from $A$ and extends the dependency relation from all body parts in $S$ to all public rules from $B$. No other dependencies between rules from $A$ and $B$ are established.

To illustrate this consider again the configuration from Figure 8. Assume that we import (1) module A into B with a scope limited to body part 3 of rule $\mathsf{R}_{B_2}$ and that (2) we import the result into the main program limiting the scope to body parts 2 and 3 of rule $\mathsf{R}_{P_1}$. Third, we import module A also directly into the main program with scope body part 1 of $\mathsf{R}_{P_1}$.

This can be compactly expressed by the following module composition expression:

$$\big(\mathsf{P} \rtimes_{(1,2),(1,3)} (\mathsf{B} \rtimes_{(2,3)} \mathsf{A})\big) \rtimes_{(1,1)} \mathsf{A}$$

As usual, such expressions are best read inside out: We import the result of importing A into B with scope $\{(2,3)\}$ into $P$ with scope $\{(1,2),(1,3)\}$ and then also import A with scope $\{(1,1)\}$. The result of this expression is shown in Figure 9.

Formally, we first introduce the concept of (module) scope.

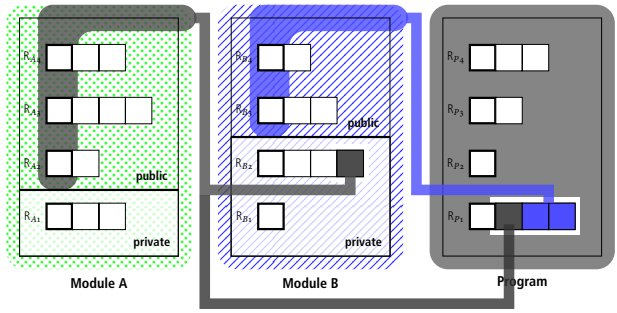



Figure 9. Scoped import of (1) module A into body part 3 of rule $R_{B_2}$ and into body part 1 of rule $R_{P_1}$ and (2) of (the expanded) module B into body part 2 and 3 of rule $R_{P_1}$. into the main program

**Definition 3.5** (Scope)**.** Let $M = (R_{\text{PRIV}}, R_{\text{PUB}}, \Delta)$ be a module (or program if $R_{\text{PUB}}$ is the empty sequence). Then a set of body parts from $M$ is called a scope in $M$. More precisely, a scope $S$ in $M$ is a set of pairs from $\mathbb{N}^2$ such that each pair identifies one rule and one body part within that rule.

For instance, the scope $\{(1,2),(1,3),(4,2)\}$ comprises for program P from Figure 8 the body parts 2 and 3 of rule 1 and the body part 2 of rule 4.

Second, we need a notation for adjusting a given dependency relation when adding rules. It turns out, a single operation (slide) suffices for our purposes:

**Definition 3.6** (Dependency slide)**.** Given a dependency relation $\Delta$, slide computes a new dependency relation by sliding all rules in the slide window $W = [s+1, s+\mathit{length}+1]$ in such a way that the slide window afterwards starts at $s_{new}+1$:

$$\begin{aligned}\mathsf{slide}(\Delta, s, \mathit{length}, s_{new}) = \{(r_1', b, r_2', h) : (r_1, b, r_2, h) \in \Delta \\ \wedge r_1' = \begin{cases} s_{new} + 1 + (r_1 - s) & \text{if } r_1 \in W \\ r_1 & \text{otherwise} \end{cases} \\ \wedge r_2' = \left.\begin{cases} s_{new} + 1 + (r_2 - s) & \text{if } r_2 \in W \\ r_1 & \text{otherwise} \end{cases}\right\}\end{aligned}$$

With this, a scoped import becomes a straightforward module composition:

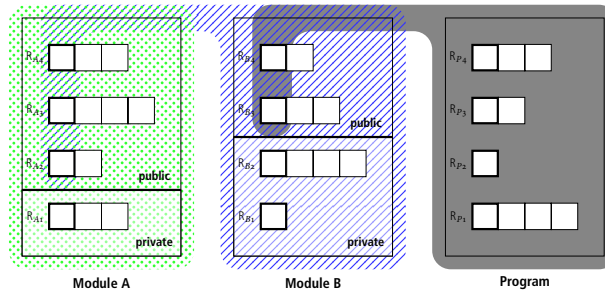



Figure 10. Private import of A into B and B into the main program

**Definition 3.7** (Scoped import $\rtimes$)**.** Let $M' = (R'_{\text{PRIV}}, R'_{\text{PUB}}, \Delta')$ and $M'' = (R''_{\text{PRIV}}, R''_{\text{PUB}}, \Delta'')$ be two modules and $S$ a scope in $M'$. Then

$$M' \rtimes_S M'' := (R_{\text{PRIV}} = R'_{\text{PRIV}} \diamond R''_{\text{PRIV}} \diamond R''_{\text{PUB}}, R_{\text{PUB}} = R'_{\text{PUB}}, \Delta'_{slided} \cup \Delta''_{slided} \cup \Delta_{inter}), \text{ where}$$

- $\Delta'_{slided} = \text{slide}(\Delta', |R'_{\text{PRIV}}|, |R'_{\text{PUB}}|, |R_{\text{PRIV}}|)$ is the dependency relation of the *importing* module $M'$ with the *public* rules slided to the very end of the rule sequence of the new module (i.e., after the rules from $M''$),
- $\Delta''_{slided} = \text{slide}(\Delta'', 1, |R''_{\text{PRIV}}| + |R''_{\text{PUB}}|, |R'_{\text{PRIV}}|)$ is the dependency relation of the *imported* module $M'$ with all its rules slided between the private and the public rules of the importing module (they have to be "between" because they are part of the private rules of the new module),
- $\Delta_{inter} = \{(r_1, b, r_2, h) : (r_1, b) \in S \wedge \exists$ a rule in $R_{\text{PRIV}}$ with index $r_2$ and head part $h$: $r_2 > |R'_{\text{PRIV}}| + |R''_{\text{PRIV}}|\}$ the inter-dependencies between rules from the importing and rules from the imported module. We simply allow each body part in the scope to depend on all the public rules of the imported module. This suffices for our purpose, but we could also choose a more precise dependency relation (e.g., by testing whether a body part can at all match with a head part).

Note, that the main difficulty in this definition is the slightly tedious management of the dependency relation when the sequence of rules changes. We, nevertheless, chose an explicit sequence notation for rule programs to emphasize that this approach is entirely capable of handling languages where the order of rules affects the semantics or evaluation and thus should be preserved.

PUBLIC AND PRIVATE IMPORT. Two more import operators suffice to make our module algebra sufficiently expressive to formalize the module system discussed in Section 3.5.1 as well as module systems for languages such as XQuery or Xcerpt.

In fact, if the language provides a (database) view concept, the single scoped import operator suffices as the two remaining ones can be defined using views on top of scoped imports. Before introducing these rewritings, we briefly introduce the public and private import operator. Formal definitions are omitted for conciseness reasons, but can be fairly straightforwardly derived from the definition for the scoped import.

All three operators hide information resulting from private rules in a module, however the public information is made accessible in different ways by each of the operators: The scoped import operator makes the information from the public interface of *B* accessible only to explicitly marked rules. The private and public import operators, in contrast, make all information from the public interface of *B* accessible to all rules of *A*. They differ only w.r.t. cascading module import, i.e., when a module *A* that imports another modules *B* is itself imported. In that case the public import operator ($\times$) makes the public interface of *B* part of the public interface of *A*, whereas the private import operator ($\bowtie$) keeps the import of *B* hidden.

Figure 10 shows the effect of the private import operator on the configuration from Figure 8 using the module composition expression $\mathsf{P} \bowtie (\mathsf{B} \bowtie \mathsf{A})$: Module B imports module A privately and the main program imports module B privately. In both cases, the immediate effect is the same: The body parts of B get access to the head parts in A's public rules and the body parts of the main program P get access to the head parts in B's public rules. The import of A into B is hidden entirely from the main program. This contrasts to the public import in the expression $\mathsf{P} \times (\mathsf{B} \times \mathsf{A})$. There the main program's body parts also depend on the head parts in A's (and not only B's) public rules.

RECURSIVE MODULE COMPOSITION. So far, we have only considered non-recursive module composition and only acyclic dependency relations. However, if the target language supports cyclic dependency relations, we can also allow recursive module composition. For that purpose we introduce a *module assignment operator* such that *module-id* := *module-expression* associates the given module identifier with the module obtained from a given module expression. Furthermore, we all module identifier to occur instead of any module composition expression as defined above. A module identifier is always mapped to the same module.

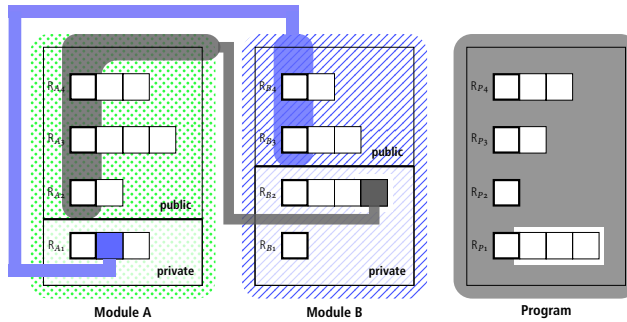



Figure 11. Scoped import of B into A with B importing A itself

For instance, the following import expression

$$M := \mathsf{A} \rtimes_{(1,1)} (\mathsf{B} \rtimes_{(2,3)} \mathsf{M})$$

yields the dependency depicted in Figure 11 where the first body part of rule 1 in $A$ depends on the public rules of $B$ which in turn depend on all rules in $B$ including rule 2 whose third body part depends on the public rules in $A$ which depend on all rules in $A$ including rule 1.

When realizing recursive module composition in Xcerpt using Reuseware [124] as described below we need to carry a mapping from module identifiers to store identifiers to ensure that multiple occurrences of the same identifier are translated with the same store identifier.

OPERATOR REWRITING. As stated above, we can express both public and private import using additional views (i.e., deductive rule) plus a scoped import, if the rule language provides view.

**Theorem 3.1** (Rewriting $\times$). *Let* $M' = (R'_{PRIV}, R'_{PUB}, \Delta')$, $M'' = (R''_{PRIV}, R''_{PUB}, \Delta'')$ *be modules and* $M = (R_{PRIV}, R_{PUB}, \Delta) = M' \times M''$. *Then,* $M^* = (R'_{PRIV}, R'_{PUB} \diamond R, \Delta') \rtimes_S M''$ *is (up to the helper predicate R) equivalent to M if*

$$R = [h \text{ :- } h : h \text{ is a head part in } R''_{PUB}]$$
$$S = \{(i,1) : |R'_{PRIV}| + |R'_{PUB}| < i \leq |R'_{PRIV}| + |R'_{PUB}| + |R|\}.$$

The gist is the introduction of "bridging" rules that are dependent on no rule in the existing module but whose body parts are the scope of the import of $M''$.

Note, that we use one rule for each public head part of the imported module. If the language provides for body parts that match any head part

(e.g., by allowing higher-order variables or treating predicate symbols like constant symbols like in RDF), this can be reduced to a single additional rule. In both cases, however, the rewriting is linear in the size of the original modules.

For private import, an analogous corollary holds (where the "bridge" rules are placed into the private rules of the module and the dependency relation properly adapted).

### 3.5.4 MODULES FOR XCERPT

Using the formal foundation for modules in rule languages, we show in the remainder of this section, how to add modules to Xcerpt using the above framework to provide the semantics for the proposed module extension. First, we illustrate the ideas along a use case, then we define the precise language constructs and show how to realize the module extension by reduction to module-free Xcerpt and observe that the semantics of that realization is exactly as prescribed by the above module system.

*Use case: Music aggregation with the Web Music Library*

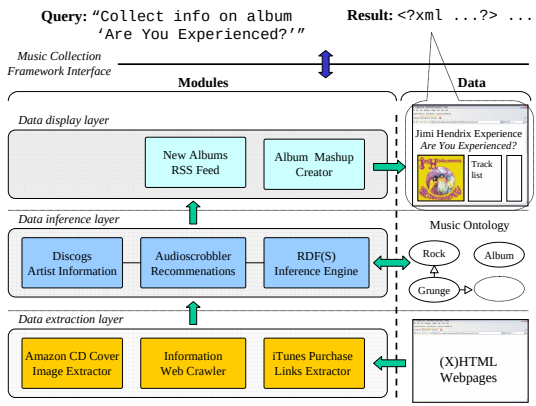



Figure 12. Many query languages only allow writing monolithic queries, while modular query development greatly increases reuse and ease of programming.

The use case illustrated in Figure 12 presents a library (called MusicLibrary) of functionality useful for coping with music and information about music found on the (Semantic) Web. At an (arguably) lower layer, information is extracted from various established Web sites like `amazon.com` or

discogs.org. The extraction has to be handled differently for every web site, but is valuable for many users and applications. For example, many of the currently established desktop music players exploit the album or CD images of Amazon to display cover art while playing back music. Encapsulating reusable queries dealing with a particular information source allow for flexible maintenance and propagation to a larger user base. The legacy information as found on external Web sites is then converted to an internal representation loosely based on the Music Ontology [105]. Music Ontology is an RDFS-based standard, hence knowledge inference and reasoning on—possibly incomplete—Music Ontology data can be achieved using an RDFS reasoner. Since such a reasoner is usable in many different fields of applications, it is implemented and provided as an Xcerpt module and included in the main library, hence allowing for its reuse. Perhaps more interesting to the end user, various modules providing pleasant visualizations of gathered information or predefined query skeletons can be provided in the library. Such modules can also be provided by third parties or, last but not least, as part of an application using the Web Music Library. We show only small extracts of the actual modules for space and presentation reasons.

*Realizing Musical Modules in Xcerpt*

How can we today realize this application in Xcerpt? In the absence of modules we have to carefully craft a single query program with a considerable number of rules (well over three dozens if we follow the basic design presented below) at each step taking great care that the rules do not, by chance, interfere with each other. Furthermore, we have to update the whole query program as soon as any information source changes, since this information is hard-coded in the program.

In the presence of a module extension, the task becomes a lot less daunting: Let us start from the top with a user program that gathers information about Jimi Hendrix from all the sources described in Figure 12. For that, it relies on a module called MusicLibrary (discussed above). The library is not a mere database, it is an interface to various ways of reasoning about musical information available on the Web. To the user the complexity remains hidden. The user just poses his query to the module without caring whether the data is extensional or intensional and how it is obtained. The module system ensures that, regardless of the actual rules and their distribution between modules, there is no chance for interference by rules of different sub-modules used within MusicLibrary.

```
1 IMPORT "MusicLibrary"
```

```
GOAL
  html [ body [
    h1 [ "Records by Jimi Hendrix" ],
    table [ tr [ td [ "Record" ], td [ "Year" ] ],
          all tr [ td [ var R ], td [ var Y ] ] ]
  ] ]
FROM
  in "MusicLibrary" (
    desc record { artist [ "Jimi Hendrix" ],
                  title [ var R ], year[ var Y ] } )
END
```

The MusicLibrary module itself is integrating data and knowledge of other modules the same way as the user program. It has to provide the information, and only the desired information, to the user of the module. Some rules may be necessary internally in the module to achieve the task, but should not be directly visible to the user of the module. The visible parts of the module are hence *public*, the others (implicitly) *private*.

Apart from using knowledge of other modules, modules may also receive data provided by importing modules. MusicLibrary accesses data extracted by a module gathering MusicBrainz metadata, feeds it to a module for converting that data to Music Ontology knowledge (Musicbrainz2MOFacts), and finally injects that knowledge to an RDFS reasoner (using the MO-Ontology-Reasoner module). It also accesses `discogs.org` directly and feeds the acquired data into another instance of the MO-Ontology-Reasoner. To distinguish multiple instances of the reasoner, each instance is given an alias (using the `@` construct), which can be used the same way as the module identifier when querying, or sending data to, a module. In this way, modules also give rise to *scoped* reasoning where consequences only apply in a certain scope (or module), but are not (automatically) propagated outside of that scope. In particular, knowledge in different scopes may, if considered globally, be inconsistent, but within each scope be consistent.

```
MODULE "MusicLibrary"
IMPORT "MusicBrainz"
IMPORT "Musicbrainz2MOFacts"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-musicBrains"
IMPORT "MO-Ontology-Reasoner" @ "reasoner-for-discogs"

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-musicBrains" ( var KNOWLEDGE ) END

CONSTRUCT public var KNOWLEDGE
FROM in "reasoner-for-discogs" ( var KNOWLEDGE ) END
```

```
CONSTRUCT to "reasoner-for-musicBrains" ( var FACTS )
FROM in "Musicbrainz2MOFacts" ( var FACTS ) END

CONSTRUCT to "Musicbrainz2MOFacts" ( var METADATA )
FROM in "MusicBrainz"( metadata [[ var METADATA ]] ) END
...
CONSTRUCT discogs-document-for-crawler[ all HREF ]
FROM in document(iri="http://www.discogs.org") ( desc a [[ href
    [ var HREF ] ]] ) END
```

Finally, let us glance at the MO-Ontology-Reasoner module which is one of the modules that not only extracts data but is injected with data to operate on. Hence, one of the queries is adorned with the **public** keyword, indicating that chaining is to be performed against the rules of the importing module that pass input data to the reasoner. Those facts, together with the ontology definition (and any domain dependent reasoning we would like to perform on the music ontology data) are sent to an RDFS reasoner module, whose consequences are then made publicly available. This RDFS reasoner is an example of a highly reusable module that can be shared among many different modules. It implements the RDF semantic in the (graph-based) query language Xcerpt (cf. [49] for details).

```
MODULE "MO-Ontology-Reasoner"
IMPORT "RDFS-Reasoner"

CONSTRUCT public var KNOWLEDGE
FROM in "RDFS-Reasoner" ( var KNOWLEDGE ) END

CONSTRUCT to "RDFS-Reasoner" ( var FACTS )
FROM public var FACTS END

CONSTRUCT to "RDFS-Reasoner" ( var MO )
FROM in document(type="xmlrdf"
    iri="http://purl.org/ontology/mo/") ( var MO ) END
```

### 3.5.5 MODULAR XCERPT—REQUIREMENTS AND CONSTRUCTS

We have seen that modules can greatly ease the development of complex Web queries (as observed increasingly) and how to apply them in examples. Before we discuss the principles of the semantics in Section 3.5.5, let us first summarize the core concepts and constructs introduced. We divide the presentation of the concepts in two parts: from the perspective of the module programmer and of the module user.

**Module programmers** need constructs for defining sets of rules and ways

of declaring appropriate access to the module—interfaces for proper encapsulation. To allow module authors to encapsulate modules, *visibility constructs* are employed. For each rule of the module, the construct term and the query term (if present) is associated with a visibility concept: *public* or *private*. Only public visibility is specifically specified, otherwise the default visibility *private* is used to encourage encapsulation.

MODULE DECLARATION: We can group sets of rules into modules and give such a set an identifier. This module can than be imported into other modules or programs.

⟨*module*⟩ ::= 'MODULE' ⟨*module-id*⟩ ⟨*import*⟩* ⟨*rules*⟩*

MODULE INTERFACES: We can declare allowed access points to a module to facilitate encapsulation and proper interfaces. Any construct term can be annotated with **`public`** to indicate that it can be queried by importing modules (see below).

⟨*interface-out*⟩ ::= 'public' ⟨*construct-term*⟩

Conversely, importing modules may provision data to an imported module (see 'module provision' below). This data is exclusively queried by query terms marked with **`public`** in the imported module.

⟨*interface-in*⟩ ::= 'public' ⟨*query-term*⟩

In other words, a module programmer defines the name and the in- and output interfaces of a module. The input of a module is accessed or queried by public query terms, the output of a module is formed by public construct terms. A module should also be complemented by documentation for the user describing its task and interfaces.

**Module users** need to be able to (a) declare which modules they want to use in a program, to (b) query the public interfaces of such modules, and to (c) provide data to such modules.

MODULE IMPORTATION: We can import modules into other modules or programs. The only effect of a module is that the module identifier (or its alias, if an alias is used) becomes available for use in module querying or provision statements. In practice, module identifiers are often rather long and complex URIs which makes the use of (short and easy to read) aliases advisable in most cases.

⟨*import*⟩ ::= 'IMPORT' ⟨*module-id*⟩ ('@' ⟨*alias-id*⟩)?

MODULE QUERYING: We can query the consequences of the public construct terms of a module. The given query term is matched only

against the results from *public* rules of the given module but neither against those from that module's *private* rules nor against other rules from the current module.

⟨*module-access*⟩ ::= 'in' ⟨*module-id*⟩ '(' ⟨*query-term*⟩ ')'

MODULE PROVISION: We can feed or provision data to the public query terms of a module. The result of a rule with such a construct term is only considered for *public* query terms in the given module, not for query terms in the current module or for query terms from the given module that are not marked *public*.

⟨*module-provision*⟩ ::= 'to' ⟨*module-id*⟩ '(' ⟨*construct-term*⟩ ')'

With only these three operations, a module user can flexibly compose modules (even multiple instances of the same module) while all the encapsulation is taken care of by the module system without further user intervention.

So far, all module access is always explicitly scoped with the module identifier. In a language with views such as Xcerpt, this suffices as we always can add a bridging rule (such as the first rule in the MusicLibrary module from Section 3.5.4) that makes all data obtained from the public interface of an imported module available to other rules in the importing module (without need for qualification). We provide two additional variants of module import for convenience that cover this case. They only differ in the way they affect module cascading: 'import public' ⟨*module-id*⟩ makes all data provided by the public interface of module *module-id* available to all unqualified rules in the importing module and also adds it to the public interface of that module whereas 'import private' ⟨*module-id*⟩ only makes it available to the unqualified rules.

*Reducing Xcerpt Modules—Stores*

The dual objectives of our approach are to (a) keep the module system simple and easy to use and to (b) allow the reuse of existing language tools and engines without modification. These two objectives actually go hand in hand, as a reduction semantics for modules (i.e., a semantics that is based on the semantics of the module-free language) proves to be elegant and easy to understand and naturally fulfills the second objective.

To allow users to truly think in terms of modules and make use of this abstraction, it is important to ensure proper and valid module interactivity *statically* before applying the module-unaware query engine to the involved rules. Thus, only the intended rule dependencies must be present in the merged rules—we have no way of enforcing rule separations during rule execution.

For the Xcerpt module system we ensure proper rule dependencies using the notion of **stores**. Intuitively, a store is a designated data area where data and queries are appropriately redirected to adhere to the proper access of rules as specified by the module programmer. A store is associated with an identifier and consists of a *private*, *in* and *out* part. Intuitively, the *private* part is intended for data access internal to the module only and the *in* and *out* parts for input and output data of that module. That is, data to be processed by the module will be injected into the *in* part of the store and data constructed by the module—upon request from another module—will reside in the *out* part of the store and can be queried by an importing module.

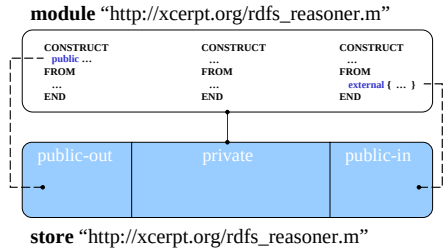



Figure 13. Module *stores* consists of three distinct areas to ensure encapsulation of data.

Stores can already be simulated using the existing Xcerpt mechanisms. Let us first assume that for each module we have one associated store that is identified by the same (unique) identifier. The construct terms and query terms of each rule in an imported module as well as rules using **in** or **to** for module access or provision in an importing module are modified such that the appropriate store is referenced:

```
in <module-id> ( <query> ) →   store [ id [ <module-id> ], access [ "out"], <query> ]
to <module-id> ( <construct> ) store [ id [ <module-id> ], access [ "in"], <construct> ]
CONSTRUCT <c> FROM <q> END →   CONSTRUCT store [ id [<module-id>], access["private"], <c>]
                               FROM store [ id [<module-id>], access["private"], <q> ] END
```

Some rules in the imported module are exempted from this transformation, viz. construct terms in goals (producing results for the end user), query terms specifically referencing an external resource (such as an XML document or other module) rather than the internal module store. Also, if the query term is a complex query it might be necessary to propagate the store specification inside the query (e.g., over disjunctions, negation, etc.).

However, these details are omitted here for space reasons.[13]

### 3.5.6 REFINING *STORES:* INSTANCE STORES

The store concept described above ensures basic encapsulation capabilities for Xcerpt modules and is attractive for its simplicity. However, there are certain situations where associating one store per module is not sufficient. Consider the situation where two modules (`A`,`B`) imports a third one (`C`) and both `A` and `B` injects data into the store associated with `C`. In such a case, after module `C` has processed the data, module `A` *may* receive data initially injected by module `B`. As such, modules `A` and `B` are not kept separate violating one of the core premises of our desire for modules. This is not a limit of the store approach, but due to the assumption of the existence of one store per module.

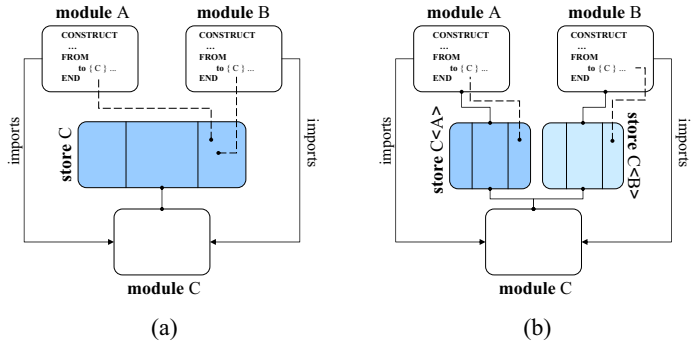



Figure 14. To improve encapsulation one store per module communication instance can be used.

To address this problem, we associate stores not with a module but with a module *import*. This can be seen as instantiating a store for each module import with the identifier of the importing module. We thus end up with two stores `C<A>` and `C<B>`, due to two import operators. A similar case where this is needed is when we use the same module but with different "feeds" using aliases. This is the case in the Music Library module presented in Section 3.5.4 where aliases (using `@`) were used to force such separations.

13 But available with examples at http://www.reuseware.org/modularxcerptexample.

*Implementation.*

Not only is it an advantage to reuse the query engine in executing the transformed and merged rules, it is also beneficial if existing technology can be used to realize the above-described transformations. To achieve this, we realize the module system via composition in the Reuseware Composition Framework [124]. The composition framework allows for the development of a light-weight composition system responsible for handling the augmented constructs related to modules. The composition framework allows both to extend the Xcerpt language with the additional syntactic constructs and to handle the transformation and merging of the involved rules in the manner described above to enforce encapsulation. The details of this implementation are left out for space reasons, but are available at `http://www.reuseware.org/modularxcerptexample`.

### 3.5.7 RELATED WORK

Despite the apparent lack of modules in many Web rule languages, module extensions for logic programming and other rule languages have been considered for a long time in research. We believe, that one of the reasons that they are still not part of the "standard repertoire" of a rule language lies in the complexity of many previous approaches.

For space reasons, we can only highlight a few selected approaches. First, in logic programming module extensions for Prolog and similar languages have fairly early been considered, cf., e.g., [41, 186, 75]. Miller [162] proposes a module extension for Prolog that includes parameterized modules similar in style to those as discussed in Section 3.5.1 and is the first to place a clear emphasis on strict (i.e., not to overcome) information hiding. In contrast to our approach, the proposed semantics requires an extension of standard logic programming (with implication in goals and rule bodies).

A reduction semantics, as used in here, is proposed in [137], though extra logical run-time support predicates are provided to allow module handling at run-time. However, the approach lacks support for module parameters and a clear semantics (most notable in the distinction between import and merge operation).

The most comprehensive treatment of modules in logic programming is presented in [47]. The proposed algebra is reminiscent of prior approaches [30] for first-order logic modules in algebraic specification formalisms (such as [202]). It shares a powerful expressiveness (far beyond our approach) and beautiful algebraic properties thanks to a full set of operators such as union, intersection, encapsulation, etc. The price, however,

is that these approaches are far more complex. We believe, that a single well-designed union (or combination) operation together with a strong reliance on views as an established and well-understood mechanism in rule languages is not only easier to grasp but also easier to realize. E.g., intersection and renaming operations as proposed in [30] can be handled by our module algebra through a combination of scoped imports and views. More recently, modules have also been considered in the context of distributed evaluation [106] which is beyond the scope of our paper.

### 3.5.8 CONCLUSIONS AND OUTLOOK

We argue that one ingredient to cope with size and diversity of information on the (Semantic) Web is *modular* query authoring and execution. We show advantages along a concrete use case dealing with music information aggregation on the Web. Furthermore, we demonstrate how it is possible to augment existing query languages—here focused on the language Xcerpt—with new constructs while reusing already developed semantics and query engines thanks to a reduction semantics approach. The proposed module system is simple to use (in contrast to many approaches from logic programming) yet provides better encapsulation and more advanced features (such as scoping and paramterization) than module systems for XSLT or XQuery.

The proposed module system has been formalized, integrated into Xcerpt 2.0, and implemented using the Reuseware Composition Framework. In further work, we would like to exploit existing techniques and tools such as Xcerpt's type system [201] for improving module composition. We are also investigating how similar techniques can be applied to add or improve module systems for other (non-rule based) query languages (for example, the module system of XSLT).

## 3.6 CONCLUSION

In the previous sections we outline how Xcerpt has evolved to even better support the vision of versatile query languages outlined in the previous section. The next chapter illustrates the use of Xcerpt for versatile Web querying along a use case proposed by the W3C and compares it to the use of multiple, specialized query languages.

# 4

# FROM XML TO RDF—W3C'S GRDDL



This chapter is a loosely based on [62].

## 4.1 INTRODUCTION

In the last years, the Semantic Web has significantly gained momentum, and the amount of RDF data on the Web has been increasing exponentially ever since the publication of the RDF recommendation. However, a great amount of such semantic data is intermingled with HTML and XML through the help of microformats[1] such as hCalendar[2], embedded RDF, and RDF/A[4].

*Introducing GRDDL*

To deal with this situation, the W3C proposes the GRDDL, *Gleaning Resource Descriptions From Dialects of Languages*, framework for the extraction of Semantic Web information from HTML and XML documents:

> *"GRDDL is a mechanism for Gleaning Resource Descriptions from Dialects of Languages. [It] introduces markup based on existing standards for declaring that an XML document includes data compatible with the Resource De-scription Framework (RDF) and for linking to algorithms (typically represented in XSLT), for extracting this data from the document." [77]*

1 http://microformats.org/

2 http://microformats.org/wiki/hcalendar

The idea behind GRDDL is to associate an XML document containing embedded RDF information with one or more transformation programs—which [77] proposes to write in XSLT [72]. These transformation programs are specifically written to extract the RDF information from the document.

*GRDDL use cases*

The W3C also published a collection of descriptions of use-cases as a motivation for employing the GRDDL method. One of these use-cases is the scheduling of a meeting between friends who publish their calendars either as hcalendar, embedded RDF[3], RDFa or RSS 1.0 on their homepages. In a first step, the XSLT-stylesheets associated with the homepages are used for harvesting the RDF information from the homepages, and the resulting RDF graphs are combined in a single RDF model. In a second step, SPARQL [183] is used to query the RDF data and find a date for the meeting that fits in everybody's schedule.

Our system implements this use case in two different manners. The first implementation follows the recommendation of the GRDDL standard, employing an XSLT processor and a SPARQL implementation (in our case Jena [4]). The second implementation uses Xcerpt [188, 51], a versatile Web and Semantic Web query language for both processing stages. The implementation of these use-cases uncovers difficulties and challenges in the authoring of GRDDL algorithms in XSLT and also in Xcerpt, and highlights advantages and disadvantages of both approaches.

## 4.2 SETTING

*hCalendar*

*hCalendar*[5] is a calendaring and events format based on the iCalendar standard[6], which is used for embedding RDF data about calendars and events in arbitrary XML—but mostly in XHTML—documents. hCalendar data typically includes information about the title, description, start, end of an event and may also specify its duration and frequency.

The following HTML fragment shows a barebone use of the hCalendar format embedded in XHTML:

```
<html><head><title>Family Calendars</title></head>
  <body> ...
    <h2>Robert's Schedule</h2>
    <div class="vevent">
      <h3 class="summary">Fashion Week</h1>
```

3 http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml

4 http://jena.sourceforge.net/

5 http://microformats.org/wiki/hcalendar

6 http://www.ietf.org/rfc/rfc2445.txt

```
      <p class="description">Well-known fashion designers will
                  present their new autumn collection</p>
      <p><span class="dtstart" title="2007-10-02T10:00Z">
         02/10/2007 from 10:00</span> until
        <abbr class="dtend" title="2007-10-02T10:30Z">
         10:30</abbr></p>
      <p>Location: <span class="location">New York</span></p>
      <p>For ticket reservation contact:
      <a class="url"
          href="mailto:nancy.wu@fashionweek.com">Nancy
          Wu</a></p>
    </div>
  <body></html>
```

The hCalendar annotations are hidden in class attributes of appropriate XHTML elements (creating "neutral" elements div and span where necessary). Thus, unless the Website author specifically requests styling for hCalendar annotations (by means of CSS stylesheets), the embedding of hCalendar annotations has no effect on the visual rendering of the data which is given in Figure 15 (where we depict calendar information with color highlights).

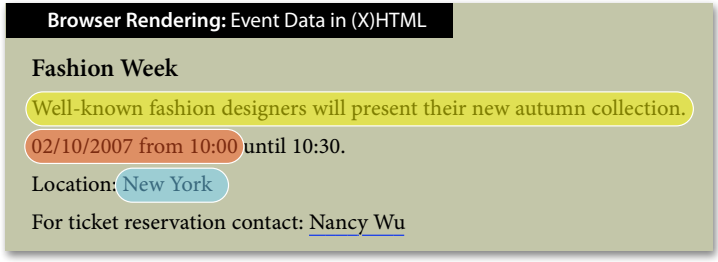



Figure 15. Browser rendering of example data

In most real-life HTML files enriched with hCalendar data found on the Web the usage of the vocabulary is more involved. In particular vevent tags may be nested within each other, and the data exhibits a very irregular structure driven by the structure of the Website's HTML markup rather than by the even structure. hCalendar is used on an increasing number of Websites, e.g., Yahoo's upcoming of O'Reilley's conference websites, cf. Figure 16.

*From XML to RDF*

Querying directly the XHTML data if we are only interested in the calendar information is fairly involved and brittle, i.e., small changes to the structure of the XHTML data that do not affect the hCalendar annotations

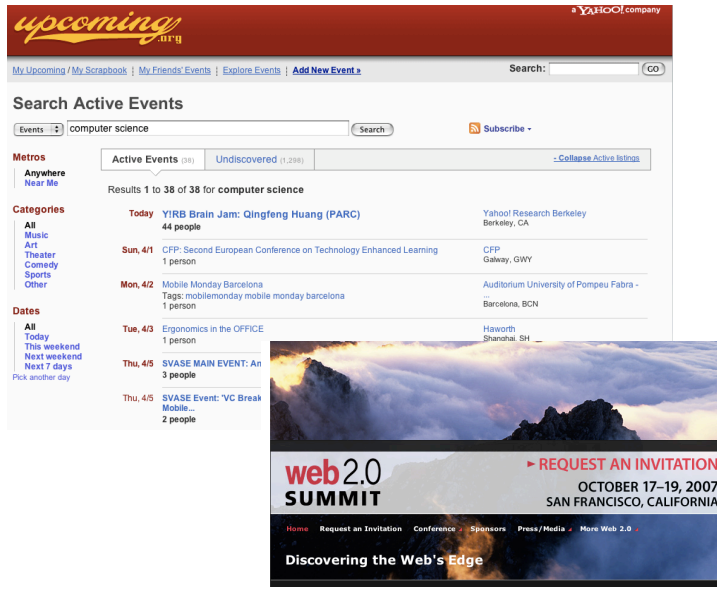



Figure 16. Exemplary website with embedded hCalendar information

may easily break queries. To avoid this, we use a view on the XHTML data that extracts only the relevant calendar information. Furthermore, we would like to be able to retrieve event data from different Web sites and process it, e.g., to "select and list all events taking place in Innsbruck during June 2007" (use-case 1), to "compare events from different sources, e.g., to find events visited by me and the colleague, I want to meet" (use-case 2), and to "analyze the event data, e.g., to find suitable free slots for a meeting" (use-case 3). We might also want to re-publish mesh-ups of the event data, e.g., on a Google Map.

All this suggests RDF as an ideal format for the view of the calendar data since RDF has been developed as integration format for data from different sources and there is already considerable calendar information that is directly represented in RDF. The RDF graph resulting from representing hCalendar data in RDF is depicted in Figure 17 and shows the full schedule of Robert and additional scheduling information for Tamara.

On this graph, we can formulate any of the above queries as if the data is originally represented in RDF.

However, the task of retrieving the RDF triples from the files is a non-trivial task. Our system follows two complementary approaches for solving this Use-Case. In the first version, it uses XSLT to transform the HTML files into RDF/XML files only including the relevant RDF data. The resulting

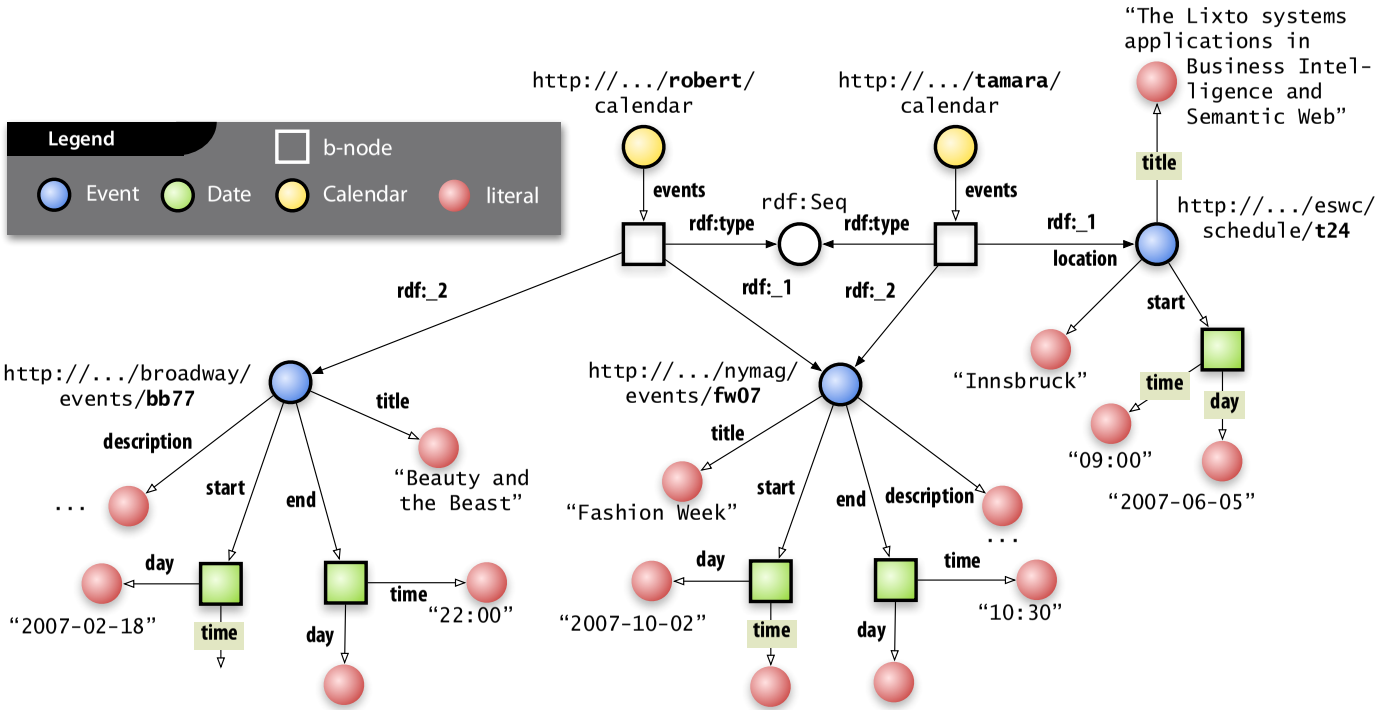



Figure 17. RDF View on hCalendar Data

RDF/XML files are then loaded into a Jena RDF repository and SPARQL is used to schedule a meeting that fits the time constraints of all participants. In the second approach, Xcerpt is used both for the extraction of the RDF triples and for scheduling the meeting.

## 4.3 FROM XML TO RDF—THE W3C WAY

*Two-stage architecture*

The GRDDL recommendation [77] suggests the use of XSLT for the transformation from XML to RDF and SPARQL for querying the resulting RDF triples. The architecture of this approach is illustrated in Figure 18.

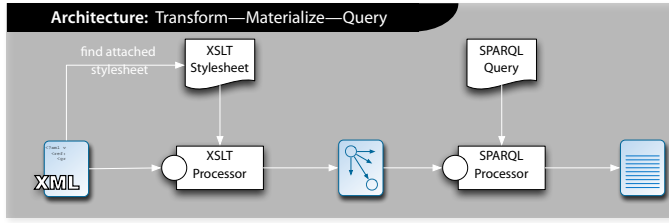



Figure 18. Architecture of W3C Approach

It is a two step processing where the XSLT processor first extracts *all* RDF triples contained in the given XML document and then, separately, the SPARQL processor answers any queries on that collection of RDF triples.

The following XSLT expression shows a brief way of finding the description for a given event (represented by an XML-element stored in the variable `$this_event`). A rather involved XPath query extracts the relevant text node which is then stored in an XSLT variable for reuse in the construction of the RDF/XML. The reason for the surprisingly complicated expression is the presence of nested event descriptions (represented by elements with vevent class attribute). A description may occur at any depth below the event element with the vevent class attribute, but only relates to the current event if there is no other event in-between the two elements.

```
<xsl:variable name="description"
  select="descendant::*[@class='description']
    [not(ancestor::*[@class='vevent']
    [ancestor::*[.=\$this_event]])]" />
```

Using conditional axis as proposed in [154], this can be expressed far more compact. To harvest complete RDF descriptions for hCalendar-embedded

events, the same has to be done for their summaries, locations, start and end dates, etc.

The transformation is further complicated by the fact that an element in XHTML may carry many class names which are then listed as a list in the class attribute, e.g., `class="vevent navigation"` to indicate that the carrying element has class names vevent and navigation.

Another challenge for the transformation from embedded hCalendar tags to RDF/XML is the conversion of the dates given in the format recommended by RFC 3339[7] to a structured representation of the date which allows for a more human-friendly querying of the RDF data with SPARQL.

*XSLT stylesheet*

The resulting XSLT stylesheet for hCalendar (which is not the most complicated microformat) thus is already far to long for inclusion here, but to give an impression consider Figure 19 where we show the full (> 500 lines) stylesheet and an excerpt of that stylesheet concerned with description and summary extraction.

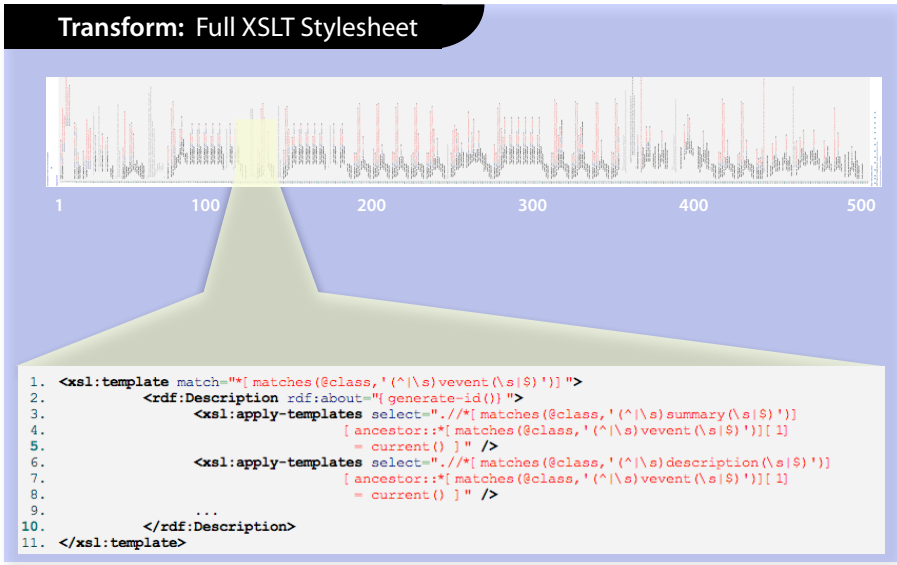



Figure 19. XSLT Transformation Stylesheet, excerpt

*SPARQL queries*

In accordance to the GRDDL use-case, our system uses the collected RDF data to find a date which fits into the schedules of all potential partic-

7 http://www.faqs.org/rfcs/rfc3339.html

ipants. SPARQL was designed to be a clean small language, which can do just enough in the context of querying RDF, without sacrificing its easy and efficient implementation. The developers of the SPARQL recommendation decided not to include negation, but negation as failure can be simulated in SPARQL with its optional triple patterns and filtering for bound variables. SPARQL is well suited for the first use-case which is easily formulated as follows:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX cal: <http://www.example.org/Calendar#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?title ?sTime
WHERE { ?x dc:title ?title. ?x a cal:Event.
  ?x cal:date ?date. ?date cal:startMonth ?sMonth.
  ?x cal:location ?location.
  OPTIONAL (?date cal:startTime ?sTime).
FILTER (?location = 'Innsbruck' AND ?sMonth = 6). }
```

Here, we select `title` and `sTime` (start time) of events with location "Innsbruck" and month 6. The start time may be missing and is then reported as **`nil`**.

The second use case can be formulated similarly. However, for the third use case, we can only give a SPARQL formulation, if we look for a specific time slot as in the following query:

```
SELECT ?title, ?x, ?y
WHERE { ?x dc:title ?title. ?x cal:date ?y.
    ?y cal:startDay ?start. ?y cal:endDay ?end.
    ?y cal:startTime ?sTime. ?y cal:endTime ?eTime
FILTER (
   ((?start = ?end && ?start = "2007-10-02" &&
     ?sTime <"12:00" && ?eTime > "11:00" )
  || (?start != ?end && ?start = "2007-10-02" &&
      ?sTime <"12:00")
  || (?start != ?end && ?end = "2007-10-02"
      && ?eTime > "11:00" )
  || (?start < "2007-10-02" && ?end > "2007-10-02"))
). }
```

## 4.4 FROM XML TO RDF—THE XCERPT WAY

In contrast to XSLT or SPARQL, Xcerpt is capable of accessing both XML (tree-shaped) and RDF (graph-shaped) data. Thus, we can implement the entire use case directly in Xcerpt. We can employ the same basic architec-

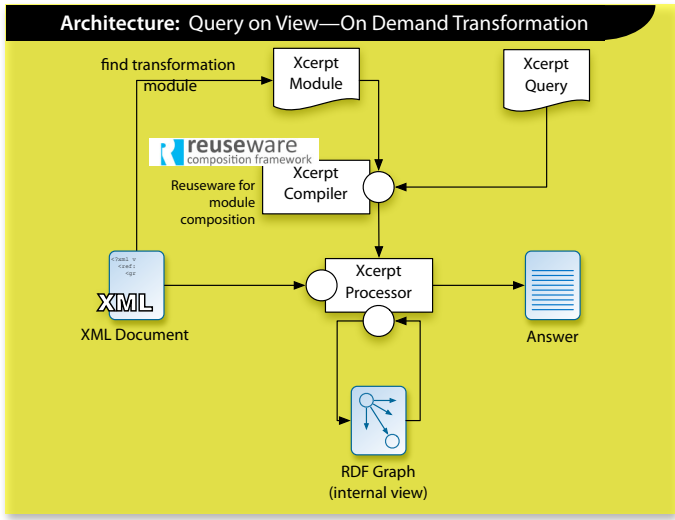



Figure 21. On-demand architecture using Xcerpt

ture as in [77], i.e., first extracting all triples, now with Xcerpt instead of XSLT, then querying the resulting graph with Xcerpt, see Figure 20.

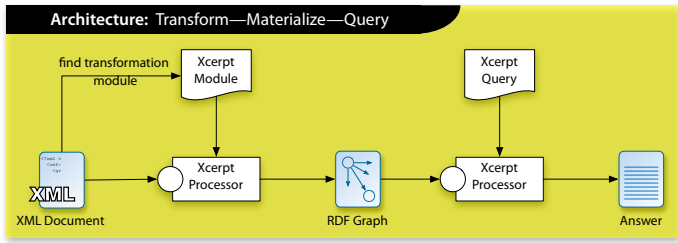



Figure 20. Two-stage architecture using Xcerpt

*Single-stage architecture*

However, with Xcerpt we can also use a single stage architecture (cf. Figure 21) where the transformation is described by Xcerpt rules that are evaluated on-demand driven by the RDF queries formulated on the result of these views. The "goal"-driven Xcerpt engine is ideally suited to ensure that only relevant parts of the XML document are actually converted (and not the entire set of triples as in the two-stage approach).

*From RDF to XML*

Providing RDF views on the calendar data is not quite as challenging as in XSLT. The basic view is as follows (_ denotes the anonymous variable

used here as label wildcard):

```
CONSTRUCT
  rdf:RDF[ all cal:Event[
     cal:summary[ var Summary ],
     cal:description[ var Description ], ... ] ]
FROM
  html {{ body {{
    desc _ ((class="vevent")) {{
      optional desc _ ((class="summary")) {
        var Summary },
      optional desc _ ((class="description")) {
        var Description },
      ... }} }} }}
END
```

However, here we do not treat nested events properly. This can be addressed by a mix of patterns and "anti"-patterns similar to the XSLT solution. However, Xcerpt 2.0 provides a more convenient way (similar to conditional axes [154] for XPath), viz. the qualified descendant as discussed in [101]. With that we can write:

```
    html {{ body {{
     desc _ ((class="vevent")) {{
       optional desc ( ! _ ((class="vevent")) )*
         _ ((class="summary")) { var Summary },
       optional desc ( ! _ ((class="vevent")) )*
         _ ((class="description")) { var Description }, ... }}
             }} }}
```

*RDF queries*

On these views, we can then easily implement all use cases discussed above: For use-case 1, we observe that Xcerpt provides a graph view of RDF rather than a triple view as in SPARQL. This is very intuitive for nested conditions as relations are immediately visible due to the query structure rather than established by variables. Furthermore, where SPARQL is limited to binding tuples or basic RDF graphs as result, Xcerpt can construct arbitrary XML or RDF data using its powerful grouping constructs. For use-case 1, the following Xcerpt rule extracts the relevant data and wraps it into a XHTML unordered list for presentation.

```
DECLARE ns-default "http://www.example.org/Calendar#"
      ns-prefix dc = "http://purl.org/dc/elements/1.1/"
      ns-prefix rdf = "http://.../22-rdf-syntax-ns#"
CONSTRUCT
  ul [ all li[ var Title " at " var sTime ] ]
FROM
  var Event {{
```

```
  --rdf:type→ Event{{ }},
  --dc:title→ literal{ var Title },
  --date→ _ {{ --startMonth→ literal{ "6" },
           optional --startTime→ literal{ var sTime } }},
  --location→ literal{ "Innsbruck" }
 }}
END
```

Note, that we use -...-> notation for traversing RDF edges. This is a syntactic refinement of the basic Xcerpt syntax further discussed in [49]. In basic Xcerpt 2.0, we obtain the same result using, e.g., `rdf:type{...}` instead of `-rdf:type-> ...`.

Finding events that both Tamara and Robert are scheduled to visit, is just as easy in Xcerpt:

```
DECLARE ns-default "http://www.example.org/Calendar#"
     ...
CONSTRUCT
 ul [ all li[ var Title ] ]
FROM
 and (
  "http://.../robert/calendar" {{
   --rdf:type→ Calendar{{ }},
   desc var Event {{
    --rdf:type→ Event{{ }},
    --dc:title→ literal{ var Title } }} }},
  "http://.../tamara/calendar" {{
   --rdf:type→ Calendar{{ }},
   desc var Event {{ }} }} )
END
```

In contrast to SPARQL, Xcerpt can find arbitrary free slots, but for comparison we show the same query as for SPARQL, viz. finding events that overlap a given time slot:

```
CONSTRUCT
 _ [ var Title, var Event, var Date ]
FROM
 var Event {{ --rdf:type→ Event{{ }},
  --dc:title→ literal{ var Title },
  --date→ var Date {{
         --startDay→ literal{ var sDay },
         --startTime→ literal{ var sTime },
         --endDay→ literal{ var eDay },
         --endTime→ literal{ var eTime } }} }}
WHERE( (sDay = eDay and sDay = "2007-10-02"
   and sTime < "12:00" and eTime > "11:00") or
```

```
13   (sDay != eDay and sDay = "2007-10-02"
       and sTime < "12:00") or
15   (sDay != eDay and eDay = "2007-10-02"
       and eTime > "11:00") or
17   (sDay < "2007-10-02" and eDay > "2007-10-02") )
   END
```

## 4.5 RELATED WORK

Although GRDDL has become recommendation only very recently several implementations of the GRDDL mechanisms already exist, and also several of its use-cases have been implemented.

The Jena GRDLL reader is a GRDDL implementation for the Jena Semantic Web framework and automatically detects and applies stylesheets referenced within HTML pages for the purpose of extraction of RDF information. In contrast to our approach, it isn't an implementation of the use case itself, but of the GRDDL mechanism. Note that there are several other implementations of the GRDDL mechanism[8].

The W3C also published an online GRDDL demo[9], which allows to extract embedded FOAF, Creative Commons, RSS, Dublin Core and GeoURL data. In contrast to our approach, it does not deal with hCalendar data and it only implements the first step of a GRDDL use case.

Dan Conolly published an XSLT stylesheet[10] for extracting hcal information from XHTML. In contrast to our system, it does not deal with nested events, does not compare alternative ways for implementation, and again only deals with the first stage of the GRDDL use-cases.

## 4.6 COMPARISON AND CONCLUSION

If we compare the two approaches, the differences are fairly obvious. The W3C approach profits from the use of standard XML and RDF query languages which are widely implemented. It is also likely, that over time a library of transformations from micro-formats to RDF triples in XSLT will become available which alleviates the burden of authoring the complex stylesheet. Nevertheless, the use of two different languages poses a considerable burden on use and deployment of this approach. Further-

8 http://esw.w3.org/topic/GrddlImplementations

9 http://www.w3.org/2003/11/rdf-in-xhtml-demo

10 http://www.w3.org/2002/12/cal/glean-hcal.xsl

more, the proposed architecture calls for a separate transformation of *all* RDF triples that can be extracted from the XML document. Though a common evaluation framework for languages such as XQuery, XSLT, and SPARQL as presented in Parts II to IV is a first step towards alleviating this problem, current implementations of XSLT or SPARQL do not allow for cross-language optimization.

In the Xcerpt case the arguments are almost inverted. We can employ a single language for the whole use case and can transform triples on-demand. Also, Xcerpt is far more expressive than SPARQL which allows us to express more interesting queries on the transformed tuples. However, Xcerpt is a non-standard query language with only prototypical implementations that is not widely adopted.

Summarizing, the presented system constitutes the first complete implementation of the GRDDL use-case and allows to draw the following conclusions: **(1)** Extracting RDF information from microformats is a non-trivial task and calls for expressive and user-friendly query languages specifically aimed at querying heterogeneous XML data. **(2)** For usability as well as efficiency purposes it is desirable to have a language that is both capable of extracting the relevant information and of further semantic processing. **(3)** Although SPARQL is a very well-specified and expected to become the most widely used RDF query language, it lacks some features—most notably grouping which limit its use in our examples. In [63] it is discussed how SPARQL can be extended with more expressive grouping constructs without increase in query complexity. Currently, these limitations of SPARQL queries mean that it must be embedded in a more powerful general purpose programming language to solve all the GRDDL use cases. **(4)** While Xcerpt is still a research prototype, it already shows that versatile, pattern-oriented and rule-based querying has the potential to considerably ease the authoring of data intensive web-applications.

GRDDL is an example of a use case, developed independently of the vision of versatile query languages and Xcerpt as discussed in the previous chapters, that illustrates the need for these approaches. It also underlines, that an evaluation framework capable of integrating different Web query languages as discussed in the remaining parts of this work is called for.

This use case concludes our glimpse at the refinement of Xcerpt's versatile aspect towards Xcerpt 2.0. In the following parts, we first discuss a novel formal foundation, called CIQLog, for Web query languages that allows us, as discussed in Part III to capture the semantics of many diverse Web query languages. Then, we discuss how queries expressed as such can be efficiently evaluated using the CIQCAG algebra and how that algebra is implemented. Though this discussion, at times, leads us quite far away from Xcerpt, we repeatedly come back to Xcerpt, when discussing the

relation of Xcerpt's data model to that of CIQLog in Section 5.5, when discussing CIQLog queries and their relation to Xcerpt or XQuery expressions in Section 6.5.3, and finally when considering the translation of (core) Xcerpt queries in Chapter 7.

Part II

# THEORY. A FORMAL PERSPECTIVE ON WEB QUERIES

# 5 DATA MODEL—RELATIONS OVER TREES AND GRAPHS



## 5.1 INTRODUCTION

Versatile languages such as Xcerpt are one avenue for addressing the fragmentation of Web data formats. In this chapter, we start with the second avenue: a uniform, purely logical semantics for many Web query languages, including versatile ones such as Xcerpt. The semantics is provided by CIQLog,

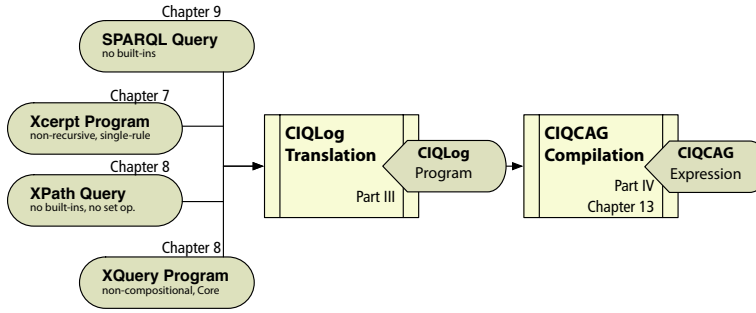



Figure 22. Overview of Parts III and IV Translation from Web Query languages to CIQLog and then to CIQCAG

a variant of datalog with negation and value invention specifically adapted to the Web setting and to be able to handle data of different shapes.

The reason for introducing CIQLog is for it to serve as the uniform semantics for many Web query languages, but also to provide a means for evaluating these languages with the CIQCAG algebra introduced in Part IV. Figure 22 illustrates this approach and relates it to parts and chapters of this thesis.

Before we turn to the translation of the individual languages, the next two chapters introduce first a uniform data model for Web data and second the language CIQLog as generalization of common Web languages.

The *data model* of CIQLog and CIQCAG (as described in Chapter 6 and Part IV) are arbitrary binary relational structures. To bridge the gap to XML and RDF query languages such as XQuery or SPARQL, we first introduce in this chapter a **common view of Web data as node and edge labeled graphs** (Section 5.2) together with mappings from XML (Section 5.3) and RDF (Section 5.4). These mappings are, for the most part, simple and intuitive. On these data graphs we define a set of unary or binary relations for querying the structure of the graph (Section 5.6.4), the relative position of edges and labels in that graph (Section 5.6.5, the labels of edges and nodes (Section 5.6.3), and edge and nodes equivalent wrt. label, position, or structure (Section 5.6.6. These relations are closely related to XPath's axes and other formalizations of relations on XML data, but exhibit a number of distinct features to properly address arbitrary shapes of the underlying data graphs and the effect of edge labels (see, e.g., the definition of child- and descendant-like traversal relations in Section 5.6.4). Such relations can then be part of binary relational structures as used by CIQLog and CIQCAG.

## 5.2 DATA GRAPHS

*Data graphs*

From the perspective of their data model, many Web representation formats such as XML, RDF, and Topic Maps have a lot of commonalities: the data is semi-structured, tree- or graph-shaped, and sometimes ordered, sometimes not (XML elements vs. XML attributes, RDF sequence containers vs. bag containers). We choose (finite unranked) **labeled ordered directed graphs** as common data model for Web data:

**Definition 5.1** (Data graph)**.** A query is evaluated against a data graph $D$ over finite label alphabets $\Sigma_E$ for edges and $\Sigma_N$ for nodes. $D$ is a 6-tuple

$$(N, E, R, \mathfrak{L}, \mathfrak{O}),$$

where $N$ is the set of nodes of the graph, $E \subset N \times \mathbb{N} \to N$ the set of edges, $R \subset N$ the set of root vertices, $\mathfrak{L} : (N \to \Sigma_N) \cup (E \to \Sigma_E)$ the labeling function on nodes and edges, and the order specification $\mathfrak{O} \subset N \times \Sigma_E$. For simplicity, we assume $N \cap E = \varnothing$ and that $\mathfrak{L}$ is total on edges, but may be partial on nodes. Note, that an edge $(n, i, m)$ maps a pair of a (source) node $n$ and an edge position $i$ to a (sink) node $m$, thus there are no two distinct edges with the same source and edge position. As usual, $\mathsf{outdeg}(n) = |\{(n, i, n') \in E\}|$ denotes the degree of a node, i.e., the number of outgoing edges.

*Order*

$D$ is an *ordered* graph, i.e., the order of the children of a node is significant. Since the order is relative to the parent and a child may occur under several parents (in fact, it may also occur several times under the same parent), the order is associated with the edge rather than with the child node. The order specification $\mathfrak{O}$ allows both ordered and unordered data (e.g., unordered XML attributes and ordered XML sub-elements, RDF bag and sequence containers) in the same graph: the order among the $\lambda$-labeled outgoing edges of a node $n$ is significant only if $(n, \lambda) \in \mathfrak{O}$. We choose to record the position of a $\lambda$-labeled edge even if $(n, \lambda) \notin \mathfrak{O}$. This allows for bag-like data with duplicates represented by multiple edges between same nodes and a consistent signature of edges.

*Node labels*

$D$'s nodes may be labeled by virtue of the single (partial) node labeling function $\mathfrak{L}$. For the sake of conciseness, we choose a single labeling function. In practice, there might be cases where different labeling functions are advisable, e.g., one for element labels and one for string values in XML data, or one for resource URIs and one for literals in RDF data.

*Edge labels*

*Edge labels* from $\Sigma_E$ are used to distinguish different relations among nodes in $N$. For an edge $e = (p, i, c)$ with $\mathfrak{L}(e) = \lambda$, we call $c$ a $\lambda$-child or simply a child of $p$. In the following, we use the edge labels CHILD and VALUE to model (element) containment and string value in an XML document.

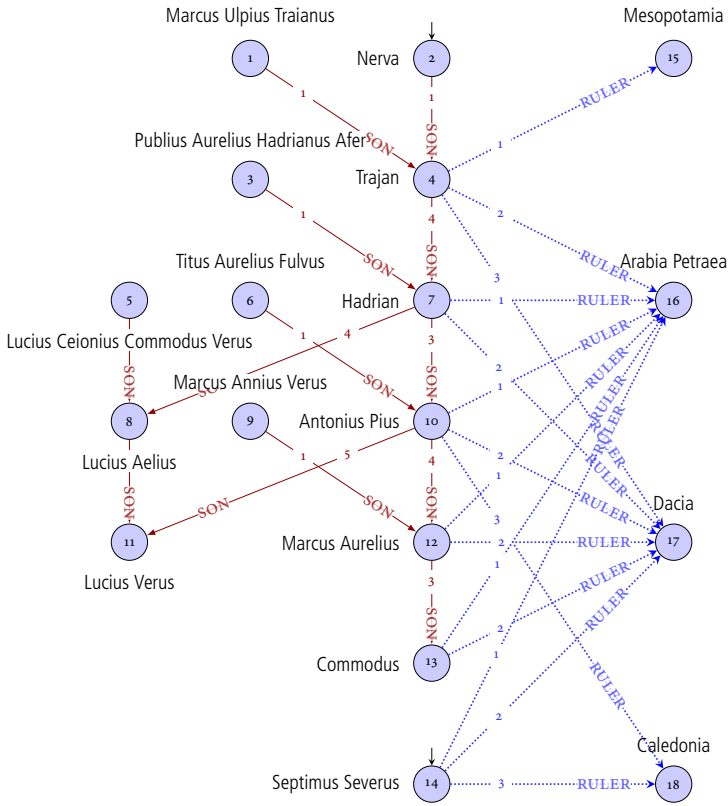



Figure 23. Exemplary Data Graph

The former represents the XML element hierarchy, the latter associates text nodes with element nodes. In case of RDF data, the respective property URI is used as edge label.

*Multiple root nodes*

The definition allows *multiple root nodes*, e.g., if there are several connected components in the graph. Any node may be a root node, in particular root nodes may, in contrast to usual rooted graph models, have parents. Intuitively, root nodes are simply highlighted "entrance" points into the graph that can be chosen arbitrarily when defining the data graph: If the data graph is a single XML document there will be a single such root node, however this formalization also covers collections of XML documents (as in XQuery) and RDF graphs where, e.g., each subject node can be considered a root node. In the following, we assume that *each node in a data graph is part of a rooted connected component*, i.e., is either a root node itself or reachable from a root node.

*Identity*

Following Codd's surrogate extension [74] for the relational data model,

we choose *surrogate or object identity for nodes and edges*, i.e., nodes and edges have identity separate from their "value" or structure. This contrasts with the basic relational data model that uses extensional identity (i.e., the value of a data item defines its identity, and thus two data items with same value necessarily have the same identity). Surrogate identity allows an intuitive and clean semantics for querying cyclic data instances, whereas cyclic data instances under extensional identity lead to infinite regular trees (cf. [80]) which have questionable properties for certain classes of queries, most prominently occurrence queries. Furthermore, [1] shows that graphs with object identity can, up to identity, be seen as finite representations of infinite regular trees. In this respect, data graphs are similar to object-oriented data models. XQuery's data model [94] also uses node identities separate from node values, but limits the data to trees.

*Example*

Figure 23 shows a data graph depicting roman emperors of the Nervan-Antonine dynasty and some of their ruled provinces. Labels are depicted in sans-serif close to the labeled node, edges are decorated with their position index (near the start of the edge, e.g., $-{}^{2}-$) and label (near the end of the edge, e.g., $\xrightarrow{\text{SON}}$), a root node is indicated by a sink-only edge (like $\rightarrow\!(2)$). The data graph contains two kinds of edges, SON edges (colored in dark red), and RULER edges (colored in blue). For referencing, nodes are numbered and we refer, e.g., to the node with label Nerva as $d_2$ (for second data graph node).

Formally, Figure 23 depicts the data graph $D = (N, E, R, \mathfrak{L}, \mathfrak{D})$ with

$$
\begin{aligned}
N &= \{d_1, \ldots, d_{18}\} \qquad\qquad R = \{d_2, d_{14}\}\\
E &= \{(d_1, 1, d_4), (d_2, 1, d_4), (d_3, 1, d_7), (d_4, 1, d_{15}), (d_4, 2, d_{16}), (d_4, 3, d_{17}),\\
&\quad (d_4, 4, d_7), (d_5, 1, d_8), (d_6, 1, d_{10}), (d_7, 1, d_{16}), (d_7, 2, d_{17}), (d_7, 3, d_{10}),\\
&\quad (d_7, 4, d_8), \ldots\}\\
\mathfrak{L} &= \{d_1 \to \text{M. Ulpius Trai.}, d_2 \to \text{Nerva}, d_3 \to \text{P. Aur. Hadr. Afer}, d_4 \to \text{Trajan}, \ldots,\\
&\quad (d_1, 1, d_4) \to \text{SON}, (d_2, 1, d_4) \to \text{SON}, (d_3, 1, d_7) \to \text{SON},\\
&\quad (d_4, 1, d_{15}) \to \text{RULER}, (d_4, 2, d_{16}) \to \text{RULER}, (d_4, 3, d_{17}) \to \text{RULER},\\
&\quad (d_4, 4, d_7) \to \text{SON}, \ldots\}\\
\mathfrak{D} &= \varnothing
\end{aligned}
$$

In the following sections, we briefly outline, how XML documents, RDF graphs, and Xcerpt data terms can all be faithfully mapped to data graphs.

## 5.3 XML: ESSENTIALS AND FORMAL REPRESENTATION

XML [43] is, by now, certainly *the* foremost data representation format for the Web and for semi-structured data in general. It has been adopted in a stupendous number of application domains, ranging from document markup (XHTML, Docbook [198]) over video annotation (MPEG 7 [152]) and music libraries (iTunes[1]) to preference files (Apple's property lists [11]), build scripts (Apache Ant[2]), and XSLT [139] stylesheets. XML is also frequently adopted for serialization of (semantically) richer data representation formats such as RDF or TopicMaps.

*XML data model defined by XML Infoset*

The following presentation of and mapping for XML documents is oriented along the XML Infoset [81] which describes the information content of an XML document. The XQuery data model [94] is, for the most parts, closely aligned with this view of XML documents.

Following the XML Infoset, we provide a *graph shaped* view of XML data containing valid ID/IDREF links. This contrasts with the XQuery data model, where such links are not resolved. In the following, ID/IDREF links are distinguished from the parent/child links expressed by the element hierarchy in accordance to the XML Infoset specification. For many applications this separation is unnecessary and even harmful which motivates us to briefly discuss an alternative mapping of ID/IDREF links to data graphs in Section 5.3.3.

### 5.3.1 XML IN 500 WORDS

*Element hierarchy*

The core provision of XML is a syntax for representing hierarchical data. Data items are called elements in XML and enclosed in start and end *tags*, both carrying the same tag names or *labels*. `<author>...</author>` is an example of such an element. In the place of '...', we can write other elements or character data as *children* of that element. The following listing shows a small XML fragment that illustrates elements and element nesting:

```
<conference xmlns:dc="http://purl.org/dc/elements/1.1/"
            dc:title="Storage Media">
  <dc:date>44 B.C.</dc:date>
  <paper title="Wax Tablets" id="p1" cites="p2">
    <author>Cicero<!-- incomplete! --></author>
  </paper>
```

1 http://www.apple.com/itunes/

2 http://ant.apache.org/

```
  <paper id="p2" cites="p1">
8   <author>Hirtius</author>
  </paper>
10 <pc><member>Cicero</member>
    <member>Atticus</member></pc>
12 </conference>
```

*Attributes: unordered*

In addition, we can observe *attributes* (name, value pairs associated with start tags) that are essentially like elements but may only contain character data, no other nested attributes or elements. Also, by definition, *element order* is significant, attribute order is not. For instance

```
<pc><member>Atticus</member><member>Cicero</member></pc>
```

represents different information than the pc element in lines 10–11, but

```
<paper cites="p1" id="p2"><author>Hirtius</author></paper>
```

represents the same element information item (inter-element white space is ignored) as lines 7–9.

*Comments, processing instructions, ...*

Elements, attributes, and character data are XML's most common information types. In addition, XML documents may also contain *comments* (line 5), *processing instructions* (name-value pair with specific semantics that can be placed anywhere an element can be placed), *document level information* (such as the XML or the document type declarations), *entities*, and *notations*. The mapping easily provides for these information types and their specifics, but details are omitted here for the sake of conciseness.

*Namespaces and base URIs*

On top of these information types, two additional facilities relevant to the mapping from XML to data graphs are introduced in XML by subsequent specifications: Namespaces [44] and Base URIs [151]. Namespaces allow the partitioning of element labels used in a document into different namespaces, identified by a URI. Thus, an element is no longer labeled with a single label but with a triple consisting of the *local name*, the *namespace prefix*, and the *namespace URI*. E.g., for the `dc:date` element in line 3, the local name is `date`, the namespace prefix is `dc`, and the namespace URI (called "name" in [81]) is `http://purl.org/dc/elements/1.1/`. The latter can be derived by looking for a *namespace declaration* for the prefix `dc`. Such a declaration is shown in line 1: `xmlns:dc="http://...` It associates the prefix `dc` with the given URI in the scope of the current element, i.e., for that element and all elements contained within unless there is another nested declaration for `dc`, in which case that declaration takes precedence. Thus, we can associate with each element a set of *in-scope namespaces*, i.e., of pairs namespace prefix and URI, that are valid in the scope of that element. Base URIs [151] are used to resolve relative URIs in an XML document. They are associated with elements using `xml:base="http://...`

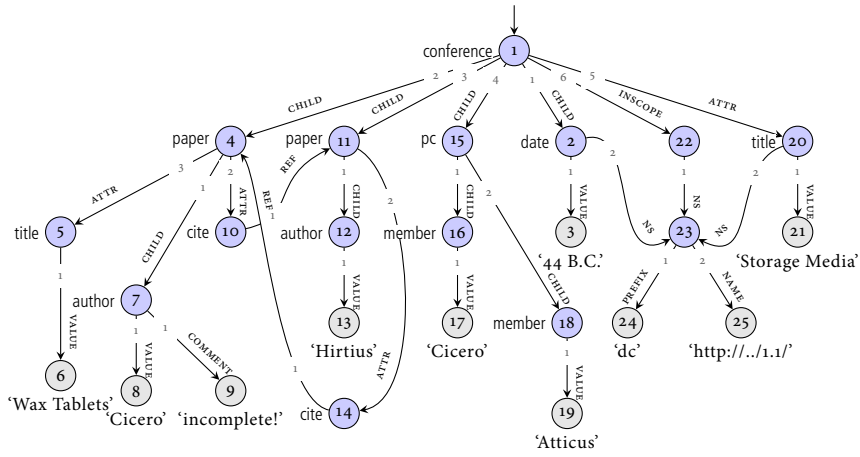



Figure 24. Exemplary Data Graph: XML Conference Data

and, as namespaces, are inherited to contained elements unless a nested `xml:base` declaration takes precedence.

In the next section, we describe how we map the basic information items as well as derived and inheritable information such as namespaces and XML Base URIs to data graphs. It is worth highlighting that the mapping can be easily extended to cover additional information items (such as entities, notations, and processing instructions or information items typed by XML Schema [92] types) as well as other forms of heritable information.

### 5.3.2 MAPPING XML TO DATA GRAPHS

*Sample XML data*

In Figure 24, a data graph for the above XML document is shown: As before, edges are decorated with their position index (e.g., $-\overset{\cdot}{\cdot}-$) and label (e.g., $\overset{\text{CHILD}}{\text{---}}$), a root node is indicated by a sink-only edge (like $\rightarrow$①). Nodes representing literal character content are differentiated using gray (like ④) instead of blue color, their labels enclosed in single quotes. The order specification is not presented in the figure. With the order specification $\mathfrak{O} = N \times \{\text{CHILD}, \text{VALUE}, \text{COMMENT}, \text{REF}\}$, i.e., order is significant for all CHILD, VALUE, COMMENT, and REF edges, but not for ATTR, INSCOPE, NS, PREFIX, and NAME edges, the graph faithfully represents the XML information set corresponding with the above fragment. As in [81], the children of an element are ordered and include element, comment, and character data children. Also, for any attribute with type idref or idrefs, the

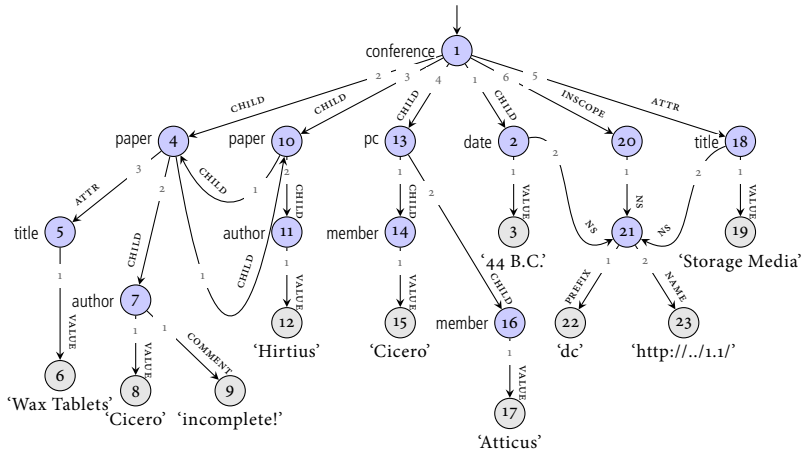



Figure 25. Exemplary Data Graph: XML Conference Data with Transparent id/idref-links

attribute information item contains an ordered list of element information items referenced by that attribute, here expressed using REF edges. This, again, mirrors [81] but deviates from the XQuery data model [94] and many other (purely tree-shaped) views of XML.

In Figure 24, we place ATTR, NS, and INSCOPE edges after CHILD, COMMENT, and VALUE edges, however, this is an arbitrary choice. As long as the order between edges of the latter kind is preserved, the order of the position of the remaining edges is insignificant.

### 5.3.3 TRANSPARENT LINKS

In the mapping scheme illustrated by Figure 24, we choose to represent id/idref-links as in [81]. However, in many applications, a treatment of such links in the same way as parent/child-relations is preferable. In this case, we place the linked elements, in order, at the beginning of the child list of the element containing the attribute.

If an element contains no other information items (no attributes, elements, comments, etc.), we may also choose to replace the element by the reference. This allows arbitrary placement of id/idref-referenced elements within the child list of a parent element. Furthermore, it allows the transparent resolution of links expressed using reference elements such as XHTML's a or Docbook's Link.

## 5.4 RDF: ESSENTIALS AND FORMAL REPRESENTATION

In addition to XML, we also demonstrate how to map RDF [150, 142, 122] graphs to datagraphs. RDF is, though much less common than XML, a widespread choice for interchanging (meta-) data together with descriptions of the schema of that data.

Following the recent SPARQL [183] proposal, we choose to support the mapping of RDF graphs under simple entailment as defined in [122]. In contrast to SPARQL, we omit typed literals and named graphs [66], both optional features of RDF (or extensions thereof) for simplicity. Both can be easily added to the described mapping, e.g., each named graph can be represented as a separate connected component, the graph representatives distinguished as root nodes of the data graph.

### 5.4.1 RDF IN 500 WORDS

RDF graphs contain simple statements about *resources* (which, in other contexts, are be called "entities", "objects", etc., i.e., elements of the domain that may partake in relations). Statements are triples consisting of subject, predicate, and object, all of which are resources. If we want to refer to a specific resource, we use (supposedly globally unique) URIs, if we want to refer to a resource for which we know that it exists and maybe some of its properties, we use *blank nodes* which play the role of existential quantifiers in logic. However, blank nodes may not occur in predicate position. Finally, for convenience, we can directly use *literal values* as objects.

RDF may be serialized in many formats (for a recent survey see [38]), such as RDF/XML [20], an XML dialect for representing RDF, or Turtle [15] which is also used in SPARQL. The following Turtle data represents roughly the same data as the XML document discussed in the previous section:

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
@prefix bib: <http://www.edutella.org/bibtex#> .
@prefix ulp: <http://example.org/roman/libraries/ulpia#> .
ulp:cicero-46-wt a bib:Article ; dc:title "Wax Tablets" ;
     dc:creator [ a rdf:Seq ;
         rdf:_1 ulp:cicero ; rdf:_2 ulp:tiro ] ;
     ulp:cites ulp:hirtius-47-bc ;
     dct:isPartOf ulp:conf-46-mutina .
ulp:cicero a bib:Person ; vcard:FN "M. T. Cicero" .
ulp:tiro a bib:Person ; vcard:FN "M. T. Tiro" .
```

```
ulp:hirtius-47-bc a bib:Article ;
14      ulp:cites ulp:cicero-46-wt ;
        dct:isPartOf ulp:conf-46-mutina .
16 ulp:conf-46-mutina a bib:InProceedings ;
        rdfs:label "Storage Media" .
```

Following the definition of namespace prefixes used in the remainder of the Turtle document (omitting common RDF namespaces), each line contains one or more statements separated by colon or semi-colon. If separated by semi-colon, the subject of the previous statement is carried over. E.g., line 6 reads as ulp:cicero-46-wt is a bib:Article and has dc:title "Wax Tablets". Lines 7–9 show a blank node: the creator of the article is neither Cicero nor Tiro, but some unnamed resource that is a sequence of those two authors.

*RDF Interpretations* are used to provide meaning to an RDF graph. URIs in subject or object position are interpreted as arbitrary objects, such as people, trains or web pages. An URI in predicate position is interpreted as a set of pairs of objects such as train connections, coauthor relationships or links between webpages. The set of resources that RDF graphs make statements about is called the *domain* of the RDF graph.

Finally blank nodes are used to express existential knowledge or to group information in RDF graphs. Each blank node is interpreted as a domain element but its interpretation is not fixed: An interpretation is a *model* of an RDF graph iff there is an interpretation for the blank nodes such that for every triple, the interpretation of the subject and object is an element of the interpretation of the predicate. An RDF graph $g$ is said to *entail* an RDF graph $h$ if every model of $g$ is also a model of $h$.

As the definition of an interpretation resembles the definition used in logic, it is possible to view an RDF graph as a formula. This formula has an atom for every triple. URIs and literals are represented by constants, while blank nodes are represented by existential variables.

### 5.4.2 MAPPING RDF TO DATA GRAPHS

*From RDF to data graphs*

This RDF data is mapped to a data graph as shown in Figure 26. In the case of RDF the order of the edges is mostly irrelevant, only for the sequence container (6 in Figure 26) we choose to order the RDF:_$i$ edges. This allows for a more convenient querying of elements in a sequence, cf. [138]. We choose to consider all named resources, i.e., all resources with URI label (depicted as light blue round nodes ◯ in Figure 26), as root nodes of the RDF graph. This choice, however, does not affect the remainder of this article. Recall, that root nodes are nothing more than specifically

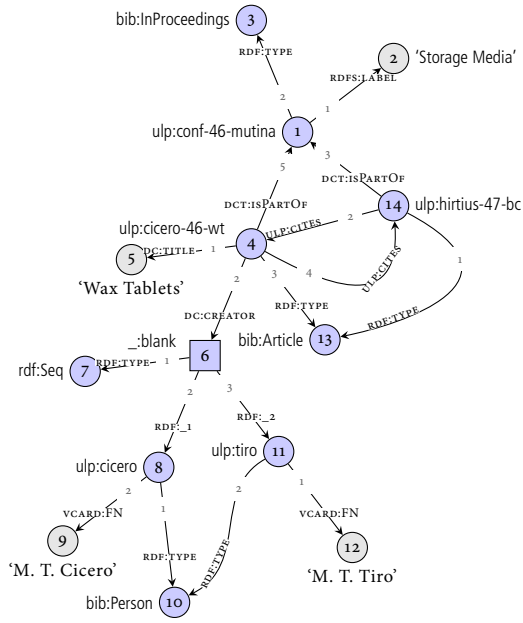



Figure 26. Exemplary Data Graph: RDF Conference Data

marked entrance points into the graph. Otherwise they are unrestricted, in particular they may have incoming edges. There is a single node in the graph for each named resource that occurs in the RDF data. The same literal may occur multiple times. Each blank node is depicted as a rectangular node (e.g., 6). As in Turtle [15] and SPARQL, blank nodes are labeled with local identifiers prefixed by `_:`. There is one node for each blank node in the RDF data.

This mapping faithfully represents the RDF graph (under simple entailment), but does not guarantee that the representation is *minimal* in the sense that there is no smaller data graph that represents an equivalent RDF graph. For an RDF graph containing blank nodes, there may be a more compact representation that eliminates some of these blank nodes that express redundant information. E.g., if we add a statement "there are two articles that cite each other" using two blank nodes for the involved article, this information is redundant as it already follows from the original data (ulp:cicero-46-wt and ulp:hirtius-47-bc cite each other). The mapping from RDF graphs to data graphs preserves such redundancy. However, the representation is minimal if the input RDF graph is minimal (or *lean* in the sense of [122]). For each RDF graph, an equivalent lean graph exists

and its computation is DP-complete[3].

Representing the actual input graph (and not an equivalent minimal one) is necessary to support queries such as selecting the blank nodes in an RDF graph (as per SPARQL's **`isBlank`** operator). Only if we *disallow queries that can distinguish blank nodes* from named resources, the lean graph returns the same answers for all queries as the original graph.

RDF provides a number of high-level modeling concepts such as collections (list), containers (bag, sequences, and alternatives), reification (the representation of a statement as a resource). There is no need to explicitly support such concepts in the mapping to data graphs, however, as all of them are reduced to certain conjunctions of basic RDF triples and thus provided implicitly. E.g., lines 7–8 in the above data show the pattern for sequence containers: A resource (often unnamed, i.e., represented by a blank node) is typed as an rdf:Seq and the elements of the sequence are connected using rdf:_*i* predicates. For details on the triple patterns for other high-level modeling concepts, see [142].

## 5.5 XCERPT DATA TERMS

We conclude our consideration of data graphs and their relation to data representation formats for the Web with a brief look at Xcerpt data terms. The formal description of the mapping is presented as part of Chapter 7. Here, we illustrate the basic ideas and how they relate to the mappings for XML and RDF data.

### 5.5.1 XCERPT DATA TERMS IN 500 WORDS

Recall from Chapter 3, the general shape of Xcerpt data terms: they are hierarchical (i.e., tree shaped) representations of *graph-shaped*, semi-structured data. To obtain a hierarchical representation of a graph, *referable term identifiers* and *references* are introduced that allow to express non-hierarchical relations. Term identifiers are like ID attributes in XML (or blank node identifiers in many RDF serializations) identifiers for data items that are unique in the context of a data collection (usually a document). References are similar to IDREF attributes in XML but occur in place of elements (rather than as attributes of a special type) and are *transparently* resolved, i.e., the case of a term containing a reference to another term cannot be distinguished from the case where the term contains the other term as a

3 Recall, that DP is the class of all decision problems, that can be expressed as the intersection of an NP- and a CO-NP-problem.

direct child.

The following Xcerpt data term yields the data graph from Figure 25:

```
declare ns-prefix dc = "http://purl.org/dc/elements/1.1/"
conference(dc:title="Storage Media") [
 dc:date [ "44 B.C." ]
 p1 @ paper(title="Wax Tablets") [
  ^p2
  author [ "Cicero"
           xcerpt:comment{ "incomplete!"} ]
 ]
 p2 @ paper [
  ^p1
  author [ "Hirtius" ]
 ]
 pc[ member[ "Cicero" ]
     member[ "Atticus" ] ]
]
```

Note that namespace declarations enclose the element (or element list) that are in the scope of that declaration, i.e., that may use the defined prefixes. Attributes are associated with term labels using parentheses, whereas the children of a term are contained in brackets or braces. Brackets indicate that the order of the children is significant, braces that it is not. The above fragment only uses brackets to yield the same data graph as an XML document where children are always ordered. However, we could just as well use braces in the above data, if, e.g., the order of papers or the order of members in a program committee is not significant.

## 5.6 RELATIONS ON DATA GRAPHS

As formal basis for the query language CIQLog and the algebra CIQCAG discussed in the following chapters, we adopt (binary) relational structures. In fact, both can be used to query arbitrary binary relational structures. For the formalization and evaluation of Web query languages such as Xcerpt, XQuery, and SPARQL, however, we choose a specific relational schema (defining the arity and names of available relations). We show how to obtain instances for this schema (i.e., the actual relations) from data graphs as described above. This bridges between the notion of data graphs that is close to the intuitive shape of semi-structured data on the Web as queried by Xcerpt, XQuery, or SPARQL, and the formal notation of relational structures that is more convenient for the purpose of defining the semantics and evaluation of CIQLog and CIQCAG.

The relations defined in the following are familiar from query languages

such as XPath and their formal treatments in, e.g., [160, 170, 112] with three major exceptions: **(1)** As we discuss *graphs* with *labeled* edges, we provide relations for accessing both nodes and edges as well as their connections where most other collections of relations on semi-structured trees or graphs either consider only nodes or only edges as the domain of the relations. **(2)** Since edges are labeled, we extend classical structural relations such as child and descendant in XPath to specify a set of edge labels to be traversed. This allows, e.g., to limit a descendant traversal to only nodes reachable by all child edges in Figure 24, excluding node reachable by a mix of child and ref edges (the latter indicating the traversal of an ID/IDREF link). The same applies to horizontal or positional relations such as following or following-sibling. **(3)** Following many object-oriented query languages as well as Xcerpt and XQuery, we introduce a deep equality on nodes such that two nodes are deep equal if the structures rooted at those nodes are, to some extent, compatible. "Compatible" may be an isomorphism between the substructures or just simulation, for details see Section 5.6.6.

Note that the domain of the attributes of these relations are either the nodes or the edges of the data graph, or the set of integers. The *active* domains are the same for nodes and edges. For integers, however, the *active* domain is a finite subset of all integers with a size bound by the maximum out-degree $d$ of a node in the data graph (and thus by the number of edges). In all cases, but pos, this set is $\{1, \ldots, d\}$. In the case of pos, the size of the set is also bound by $d$, but the contained integers are arbitrary.

### 5.6.1 BINARY RELATIONAL STRUCTURES

First, we briefly recall a (standard) definition, following [2], for relational structures but limited to binary relations:

*Data: binary relational structures*

Both CIQLOG and CIQCAG operate on a (slightly extended)[4] *relational structure* $D$ as data. $D$ is defined over a relational schema $\Sigma = (R_1[U_1], \ldots, R_k[U_k])$ and a *finite* domain $N$ of nodes (or objects or elements or records) in the data graph. Each $R_i[U_i]$ is a relation schema consisting in a relation name and a finite, nonempty set of attribute names, We assume an equality relation = on the nodes that relates each node to itself only (*identity*). $D$ is a tuple $(R_1^D, \ldots, R_k^D, O)$. Each $R_i^D$ is a finite, unary or binary relation over $N$ with name $R_i$. For a relation $R$, $\mathsf{ar}(R)$ denotes its arity. We extend $D$ with an order mapping $O$ that associates with each (binary) $R_i$ a total

4 The deviation lies in the addition of order for each relation. Furthermore, we restrict ourselves to binary relations.

order on the domain (of nodes) $N$ such that all $n \in \operatorname{rng} R_i$ are before all $n' \in N \setminus \operatorname{rng} R_i$.[5] We denote with $O(D) = \{o : \exists R_i \in D : O(R_i) = o\}$ the set of (total) orders to which the relations in $D$ are mapped. These (total) orders serve to represent the image of each node in a relation as one or more continuous intervals over the order associated with that relation. Choosing an appropriate order for a relation is discussed in Sections 11.3.1 (for tree data) and 11.3.2 (for CIG data).

### 5.6.2 A RELATIONAL SCHEMA FOR DATA GRAPHS

CIQLOG and CIQCAG operate on arbitrary relational structures, though they profit from relational structures where some or all relations are tree, forest, or CIG shaped. In the following, we outline a particular relational schema containing relations on data graphs (as introduced in Section 5.2) that is used to realize Xcerpt, XQuery, and SPARQL queries in CIQLOG (and thus CIQCAG).

Given a data graph $D = (N, E, R, \mathfrak{L}, \mathfrak{O})$ over node labels $\Sigma_N$ and edge labels $\Sigma_E$, we choose as domain $N' = N \cup E \cup \{1, \ldots, \max(\{i : \exists n, n' : (n, i, n') \in E\}) \cup \Sigma_N \cup \Sigma_E$, i.e., the union of the disjoint sets of nodes, of edges, of integers from 1 to the maximum edge position, and of node and edge labels. Note that each of the sets, and thus $N'$ too, is finite. We add node and edge labels as well as (edge position) integers to the set of nodes and edges to keep the relational schema for querying data graphs *independent* of the actual graph.[6]

Table 1 gives a summary of the relations defined in the following together with their relation schema. It is worth emphasizing that not all of these relations are used in the translation of each language. Rather, for the most part we limit ourselves to specific label sets $S$ (mostly, the singleton sets) and only consider a small number of path relations. Furthermore, not all relations must be extensional. In fact, in Chapter 6, we briefly discuss a set of minimal set of extensional relations and show how to specify the remaining relations as intensional rules on top of this minimal set in CIQLOG.

In the following, let $\Sigma = \Sigma_N \cup \Sigma_E$ be the set of node and edge labels and $\mathcal{P}(\Sigma)$ be the power set over $\Sigma$. Furthermore on a domain $D$, we denote with $R_1 \circ R_2 = \{(n, m) \in D^2 : \exists n' \in D : (n', m) \in R_1 \wedge (n, n') \in$

5 Here and in the following we denote the domain and range of a function $f : N \to M$ by $\operatorname{dom} f$ and $\operatorname{rng} f$, resp.: $\operatorname{dom} f = \{n \in N : \exists minM : f(n) = m\}$ and $\operatorname{rng} f = \{m \in M : \exists n \in N : f(n) = m\}$.

6 Alternatively, we can use one (unary) position relation for each edge position and one (unary) label test relation for each edge or node label in the data. Since both are finite sets, the resulting relations still form a relational schema.

| | node | edge | node-node | node-edge | edge-edge |
|---|---|---|---|---|---|
| identity | | | $\doteq$ | $\doteq$ | $\doteq$ |
| structure | | | $\mathsf{path}^S_{i,j}$, $\triangleq$ | $\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$ | |
| position | $@^S$ | $@^S$, pos | $\ll^S$, $\ll^S_+$, $\ll^S_*$, $\blacktriangleleft^S$, $\blacktriangleleft^S_+$, $\blacktriangleleft^S_*$ | $\prec^S$, $\prec^S_+$, $\prec^S_*{=}^S_@$ | |
| label | $\mathfrak{L}$, $\mathsf{Lab}^S$ | $\mathfrak{L}$, $\mathsf{Lab}^S$ | $\cong$ | $\cong$ | $\cong$ |
| arity | $\mathsf{indeg}^S$, $\mathsf{outdeg}^S$ | | | | |
| ordered | $\mathfrak{O}$, $\mathfrak{O}^S$ | | | | |
| root | root | | | | |

Table 1. Summary of query relations ($S$ is a set of labels from $\Sigma_N \cup \Sigma_E$)

$R_2\}$ the composition of two binary relations $R_1$ and $R_2$ and with $R^k = \underbrace{R \circ R \circ \ldots \circ R}_{k \text{ times}}$ $k$ compositions of $R$ with itself. For a binary relation $R$ let $R(n) = \{m \in N \cup E : (n,m) \in R\}$ be the "images" of $n$ under $R$.

### 5.6.3 PROPERTIES OF NODES AND EDGES: LABELS AND POSITIONS

Property relations test a certain "local" property of a node or edge without relating it to other nodes and edges. Instead, they associate an edge or node, e.g., with its label or its position among siblings or, resp., edges with the same source.

LABEL RELATION. To obtain the label of a node or edge, we make the labeling function of a data graph accessible as a binary *label relation* that identifies all edges and nodes with their label:

$$\mathfrak{L} = \{(t,\sigma) \in (N \cup E) \times \Sigma : \mathfrak{L}(t) = \sigma\}$$

For any finite set of labels $S \subset \mathcal{P}(\Sigma)$, we provide as convenience the *label test relation*

$$\mathsf{Lab}^S = \{t \in N \cup E : \mathfrak{L}(t) \in S\} = \bigcup_{\sigma \in S} \{t \in N \cup E : \mathfrak{L}(t,\sigma)\}$$

For brevity, we write for singleton sets $S = \{\lambda\}$ just $\mathsf{Lab}^\lambda$ and omit the index for $S = \Sigma$. Note, that $\mathsf{Lab} = \mathsf{Lab}^\Sigma$ is not necessarily $N \cup E$ since there may be nodes (though no edges) without label, i.e., $\mathsf{Lab}^{\Sigma_E} = E$, but $\mathsf{Lab}^{\Sigma_N}$ is not necessarily $N$. For the translations discussed in Part III, we only use

label relations with singleton label sets $S$. In this case, the number of label relations is bound by the size of $\Sigma_E \cup \Sigma_N$.

POSITION RELATION. To test for the position of an edge among the edges with the same source or of a node among its siblings, the *position relation* $@^S$ associates an edge $e$ with 1+ the number of edges with the same source that **(1)** are ordered (i.e., have a label in $\mathfrak{O}(n)$ where $n$ is the common source), **(2)** have a label in $S \in \mathcal{P}(\Sigma)$, and **(3)** have an edge position preceding the edge position of $e$. We increment the number of edges by 1 to achieve node positions between 1 and $\mathsf{outdeg}(n)$ where $n$ is the source of an edge (rather than 0 and $\mathsf{outdeg}(n-1)$). For nodes, $@^S$ associates a node $n$ with whatever any edge with $n$ as sink is associated with under $@^S$:

$$\begin{aligned} @^S = \{(e, i+1) \in E \times \mathbb{N} : e = (n,k,m) \wedge \mathfrak{L}(e) \in \mathfrak{O}(n) \\ \wedge \left|\{e' = (n,l,m') \in E : \mathfrak{L}(e') \in S \cap \mathfrak{O}(n) \wedge l < k\}\right| = i\} \\ \cup \{(n,i) \in N \times \mathbb{N} : \exists e \in E : e = (n',k,n) \wedge @^S(e,i)\} \end{aligned}$$

Notice, that an edge in $@^S$ is not itself required to be labeled with a label in $S$. Rather, this can be achieved by an additional label relation on the edge. Note, that $@^S$ is a function (for each $S$) on edges, but for a node there may, in general, be several positions associated with it: This is the case for any node with multiple incoming edges. On tree data, $@^S$ is a function also on nodes. Again we write only $\lambda$ for singleton sets $S = \{\lambda\}$ and omit the index $S$ for $S = \Sigma$. For the translations discussed in Part III, we only use position relations with singleton label sets $S$. In this case, the number of position relations is bound by the size of $\Sigma_E \cup \Sigma_N$.

For edges, we also provide a relation to retrieve the actual edge position:

$$\mathsf{pos} = \{(e,i) \in E \times \mathbb{N} : \exists n, n' \in N : e = (n,i,n')\}$$

DEGREE RELATIONS. To test for the number of in- or out-edges of a node, there are, for each $S \in \mathcal{P}(\Sigma)$, *in- and out-degree relations* $\mathsf{indeg}^S$ and $\mathsf{outdeg}^S$ that associate each node with its in- resp. out-degree, counting only edges labeled from $S$:

$$\begin{aligned} \mathsf{indeg}^S &= \{(c,i) \in N \times \mathbb{N}\} : \left|\{e = (p,j,c) \in E : \mathfrak{L}(e) \in S\}\right| = i\} \\ \mathsf{outdeg}^S &= \{(p,i) \in N \times \mathbb{N}\} : \left|\{e = (p,j,c) \in E : \mathfrak{L}(e) \in S\}\right| = i\} \end{aligned}$$

Notice, that the degree is defined over the number of edges labeled from $S$, not the number of child nodes reached over edges labeled from $S$. If the graph is simple, i.e., contains no multi-edges, both definitions are equivalent. But in the presence of multi-edges the given definition leads

to more intuitive (and higher) degrees than a definition based on child nodes. The same abbreviations as above for singleton sets and $S = \Sigma$ are used.

ORDERED RELATION. For any finite set of labels $S \in \mathcal{P}(\Sigma)$, we provide as convenience the *order specification test relation*

$$\mathfrak{O}^S = \{n \in N : \forall \sigma \in S : \mathfrak{O}(n, s)\}$$

The order specification $\mathfrak{O}$ is exposed directly.

ROOT RELATION. Finally, there is a *root node relation* $\mathsf{root} = R \subset N$ that identifies all root nodes in the data graph.

### 5.6.4 STRUCTURAL RELATIONS

The primitives for traversing the structure of the graph data are relations that connect nodes with incident edges or nodes with other nodes reachable via (arbitrary or fixed length) paths.

SOURCE AND SINK RELATIONS. Traversal from edge to node and vice versa is achieved using the *source* and *sink relations*.

$$\circ\!\!\rightarrow = \{(n, e) \in N \times E : \exists i \in \mathbb{N}, n' \in N : (n, i, n') = e\}$$
$$\rightarrow\!\!\circ = \{(n, e) \in N \times E : \exists i \in \mathbb{N}, n' \in N : (n', i, n) = e\}$$

$\rightarrow\!\!\circ$ and $\circ\!\!\rightarrow$ relate a node to *all* its in- resp. out-edges. To retrieve only one particular edge, e.g., the $i$-th out-edge, a combination of $\circ\!\!\rightarrow$ and @ can be used. To retrieve edges with a specific edge label $\circ\!\!\rightarrow$ can be combined with $\mathsf{Lab}$.

PATH RELATIONS. Structural node-node relations relate nodes that are connected by fixed length or arbitrary length paths (i.e., sequences of edges). For any $S \in \mathcal{P}(\Sigma)$, $i, j \in \mathbb{N} \cup \{\infty\}$ we define the *path relation*

$$\begin{aligned}\mathsf{path}^S_{i,j} = \{(n, m) \in N^2 : \exists i \leq k \leq j, e_1, \ldots, e_k \in E, n_1, \ldots, n_{k-1} \in N : \\ \circ\!\!\rightarrow^S(n, e_1) \wedge \rightarrow\!\!\circ^S(m, e_k) \wedge \mathsf{Lab}^S(e_k) \wedge \\ \bigwedge_{l=1}^{k-1} (\rightarrow\!\!\circ^S(n_l, e_l) \wedge \circ\!\!\rightarrow^S(n_l, e_{l+1}) \wedge \mathsf{Lab}^S(e_l))\}\end{aligned}$$

Intuitively, two nodes are in $\mathsf{path}^S_{i,j}$ relation, if there is a path with length between $i$ and $j$ that connects the two nodes. For $i = j = 1$, this renders the direct edge (or child) relation between nodes. On XML data

represented as in Figure 24 this renders, for $i = 0, j = \infty$ and $S = \{\text{CHILD}, \text{COMMENT}, \text{VALUE}\}$, XPath's descendant-or-self, for $i = 1, j = \infty$, XPath's descendant or Xcerpt's desc. For the translations discussed in Part III, we only use those three types of path (child, descendant, and descendant-or-self).

If $i = j = 1$, we omit the interval index, if $i = 0, j = \infty$ we use $*$, if $i = 1, j = \infty$ we use $+$ as index. If $S = \Sigma_E$ we omit the label index, if $S = \{\lambda\}$ we write $\lambda_{i,j}$ for $\mathsf{path}_{i,j}^{\{\lambda\}}$.

### 5.6.5 ORDER RELATIONS

The following relations are successor and order relations on edges and nodes based on the relative position of the edges or nodes under a common parent or ancestor. For each type, there is a (non-transitive) successor relation and a transitive order relation. The order relations are proper *(strict partial) orders* only for edges, for nodes (due to multi-edges, i.e., several edges with same source and sink but different edge position) the relations are strict *preorders*, i.e., not anti-symmetric. E.g., if $a$ is connected by the first and sixth edge to $b$ and by the third edge to $c$, $c$ is both a following sibling ($b \ll_+ c$) of $b$ and its preceding sibling ($c \ll_+ b$).

ORDER RELATIONS ON EDGES. Since graphs may carry order, we can compare two edges wrt. their relative position within the out-edges of a common parent. For all $S \in \mathcal{P}(\Sigma)$, we define the *direct* $\prec^S$, the *transitive* $\prec_+^S$, and the *transitive reflexive edge sibling relation* $\prec_*^S$:

$$\begin{aligned}
\prec^S &= \{(e, e') \in E^2 : \circ\!\!\rightarrow(n, e) \wedge \circ\!\!\rightarrow(n, e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&\qquad @^S(e, i) \wedge @^S(e', i+1)\} \\
\prec_+^S &= \{(e, e') \in E^2 : \circ\!\!\rightarrow(n, e) \wedge \circ\!\!\rightarrow(n, e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&\qquad @^S(e, i) \wedge @^S(e', j) \wedge i < j\} \\
\prec_*^S &= \{(e, e') \in E^2 : \circ\!\!\rightarrow(n, e) \wedge \circ\!\!\rightarrow(n, e') \wedge \mathsf{Lab}^S(e) \wedge \mathsf{Lab}^S(e') \wedge \\
&\qquad @^S(e, i) \wedge @^S(e', j) \wedge i \leq j\}
\end{aligned}$$

Intuitively, $\prec^S$ relates (under the common parent node) each edge to its immediate successors among the edges with label in $S \cap \mathfrak{O}(n)$. It is an injective function even in presence of multi-edges. $\prec_+^S$ relates (under a common parent node $p$) each node to all following edges outgoing from $p$ with label in $S \cap \mathfrak{O}(n)$. As in the case of $@^S$, the labels $S$ are only used to limit the considered edges.

ORDER RELATIONS ON NODES. There are two types of positional relations on nodes: those relating siblings under a common parent and their generalization to the entire graph, i.e., positional relations relating arbitrary nodes in a given graph.

For any $S \in \mathcal{P}(\Sigma)$, there is a *direct* $\ll^S$, a *transitive* $\ll^S_+$, and a *transitive reflexive node sibling relation* $\ll^S_*$.

$$\ll^S = \{(n,m) \in N^2 : \rightarrow\!\circ(n,e) \wedge \rightarrow\!\circ(m,e') \wedge e \prec^S e'\}$$
$$\ll^S_+ = \{(n,m) \in N^2 : \rightarrow\!\circ(n,e) \wedge \rightarrow\!\circ(m,e') \wedge e \prec^S_+ e'\}$$
$$\ll^S_* = \{(n,m) \in N^2 : \rightarrow\!\circ(n,e) \wedge \rightarrow\!\circ(m,e') \wedge e \prec^S_* e'\}$$

It follows from the definition of $\prec^S$, that two nodes in $\ll^S$ must have a common parent. On nodes, $\ll^S$ is a function only on trees. Again the same abbreviations as above for singleton sets and $S = \Sigma$ are used.

For all $S \in \mathcal{P}(\Sigma)$, there is also a *transitive* $\blacktriangleleft^S_+$, a *transitive reflexive* $\blacktriangleleft^S_*$, and a *direct following relation* $\blacktriangleleft^S$ on nodes.

$$\blacktriangleleft^S_+ = \{(n,m) \in N^2 : n' \ll^S_+ m' \wedge \mathsf{path}^S_*(n',n) \wedge \mathsf{path}^S_*(m',m)\}$$
$$\blacktriangleleft^S_* = \{(n,m) \in N^2 : n' \ll^S_* m' \wedge \mathsf{path}^S_*(n',n) \wedge \mathsf{path}^S_*(m',m)\}$$
$$\blacktriangleleft^S = \{(n,m) \in N^2 : n \blacktriangleleft^S_+ m \wedge \nexists n' \in N : n \blacktriangleleft^S_+ n' \wedge n' \blacktriangleleft^S_+ m \vee \mathsf{path}^S_*(n',n)\}$$

Intuitively, $\blacktriangleleft^S_+$ relates $n$ to all nodes $m$ that follow $n$ within the subgraph induced by all edges with label in $S$. Notice that, $\blacktriangleleft^S_+(x,y) \implies \exists y' \blacktriangleleft^S (x,y')$ only if $\blacktriangleleft^S_+$ is acyclic and thus irreflexive. If $\blacktriangleleft^S_+$ is cyclic, there is no direct following for all nodes on the cycle.

On XML data represented as in Figure 24, $\ll^{\text{CHILD}}_+$ represents XPath's following-sibling, $\blacktriangleleft^{\text{CHILD}}_+$ XPath's following axis.

### 5.6.6 EQUIVALENCE RELATIONS

The previous binary relations relate nodes and edges based on how they are connected in the data graph. Equally important is the ability to relate two nodes based on their local properties, e.g., their label, structure, or arity. In this section, we introduce several *equivalence* relations based on such properties.

LABEL EQUIVALENCE RELATION. The *label equivalence* relation $\cong$ relates all nodes or edges that have the same label.

$$\cong = \{(t_1,t_2) \in (N \cup E)^2 : \exists \lambda \in \Sigma : \mathfrak{L}(t_1) = \lambda = \mathfrak{L}(t_2)\}$$

IDENTITY EQUIVALENCE RELATION. The *identity equivalence* relation $\doteq$ relates each node or edge to itself and itself only.

$$\doteq = \{(t, t) \in (N \cup E)^2\}$$

POSITION EQUIVALENCE RELATION. For any $S \in \mathcal{P}(\Sigma)$, the *position equivalence* relation $=^S_{@}$ relates all edges that occur at the same position among the out-edges of their respective sources.

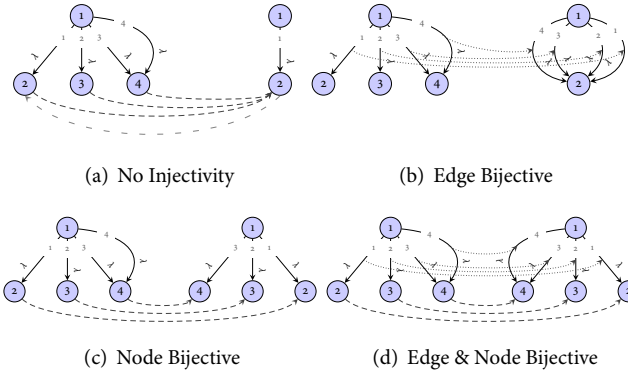
$$=^S_{@} = \{(e, e') \in E^2 : \exists i \in \mathbb{N} : @^S(e, i) \wedge @^S(e', i)\}$$

Again we use only $\lambda$ for singleton sets $S = \{\lambda\}$ and omit the index $S$ for $S = \Sigma$.

STRUCTURAL EQUIVALENCE RELATION. As the label equivalence relation stand to the label relation, structural equivalence relations stand to edge and position relations. They relate nodes not based on the equivalence of their local properties, but on the equivalence of their structure, i.e., of the subgraph rooted at the respective node. Unfortunately, since they are dealing in equivalence not between atomic values like label equivalence relations but between structured values, their semantics and evaluation is considerably more complex. In the following, we introduce a flexible structural equivalence relation, referred to following common notation as deep equal. The introduced relation is flexible enough to cover a large set of existing or desirable specific structural equivalence relations:

XQuery, e.g., provides the deep-equal function that identifies pairs of nodes with a structure that is equivalent w.r.t. the XQuery data model (thus, e.g., disregarding attribute ordering and in-scope namespaces). [144] shows that the presence of deep-equal does not affect the complexity of composition-free XQuery. Recall, that composition-free XQuery is a restriction of XQuery similar to the algebra discussed in this work, where the domain of all query operators is limited to the input document. However, XQuery operates only on ordered tree data, where deep-equal (i.e., ordered tree isomorphism) is linear [6]. The generalization of XQuery's deep-equal to general unordered graphs, however, subsumes to graph isomorphism which is believed not to be in P and which exhibits, for the general case, only exponential-time deterministic algorithms [143]. However, there exist efficient algorithm for rather large classes of graphs, e.g., planar graphs [126].

Moreover different queries might require different notions of deep join: order may be significant or insignificant, certain edges or nodes (e.g., representing comments in XML) may be entirely ignored, and non-injective mappings may be acceptable to establish equivalence. From what occurs in



(a) No Injectivity (b) Edge Bijective

(c) Node Bijective (d) Edge & Node Bijective

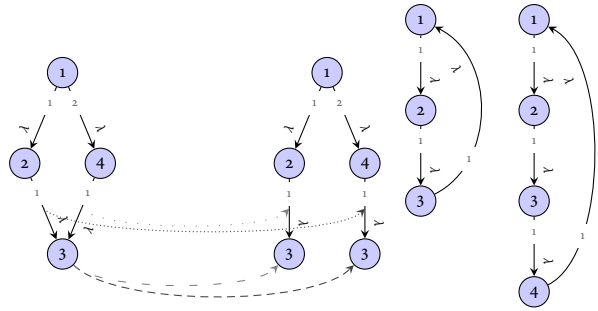
Figure 27. deep equal: effect of injectivity

practical XML and RDF query languages, these variances can be classified in three dimensions:

**(1)** WHAT SHOULD BE MAPPED BIJECTIVELY? In some cases, two nodes are considered equal already if all the structural information from one node occurs in some form in the other and vice versa. It is not required that multiple occurrences of same information is carried over. This roughly corresponds to simulation [164] as equivalence relation. On the other extreme, one might consider two nodes equivalent only when they are fully isomorph. This is, e.g., the semantics of XQuery's deep-equal.
What makes these cases different is whether the mapping between the two nodes and their respective substructures is bijective or not. More precisely, one can distinguish deep equals by the "degree" of bijectivity required:

**(a)** *Type cover:* The first choice lies in whether we demand that the mapping is (i) not bijective at all, (ii) bijective only on edges, (iii) bijective only on nodes, or (iv) bijective on both. On trees, case (ii) to (iv) are obviously equivalent, but in presence of multi-edges or cycles differences emerge. Figure 27 illustrates the differences between the different forms.

**(b)** *Structural cover:* Aside from the question which types are covered by the mapping, a further variance lies in the structural extent of the cover: either (i) the entire (reachable) subgraphs rooted at the two nodes, (ii) only adjacent edges and nodes, or (iii) only outgoing edges and children. In trees all three forms are equivalent, but in graphs the latter two are less restrictive than the first: Figure (a) shows an example of two graphs that are equivalent under (iii), but not under (i) or (ii). In general, (iii) can not distinguish DAGs from trees in all



(a) Multi-parent nodes (b) Cycle length

Figure 28. deep equal: effect of cover for equivalence mapping

cases. Figure (b) shows a case where (ii) considers the two graphs equivalent, but (i) does not. In general, (ii) fails to distinguish graphs with cycles differing only in cycle length.

**(2)** WHAT SHOULD BE MAPPED AT ALL? Some parts of the structure may be excluded, e.g., comments in XML data or annotation properties (such as rdfs:seeAlso or rdfs:label) in RD data. The deep equal presented below allows to limit the mapping to certain given edge labels $L$.

**(3)** WHAT SHOULD BE MAPPED IN ORDER? In some cases, one might be interested in preserving the order of the data (if it is ordered at all). However, in other cases (in particular for non-bijective mappings) the order may be irrelevant. As in the previous case, the deep equal presented here allows to limit the order-respecting edge labels $L_O \subset L$.

Figure 29 shows the formal definition of the generalized deep equal relation $\triangleq$ (omitting analogous variant 2b): We define four variants of $\triangleq$ that differ, as explained above, in what nodes and edges are mapped and whether the mapping is bijective. The basic case, $\triangleq$, maps only adjacent nodes and edges (more precisely, eds are edge positions in the context of each node). Its bijective variant, $\triangleq_{bij}$, also only maps adjacent nodes and edges, but does so using a bijective mapping. Analog variants, $\triangleq_*$ and $\triangleq_{*,bij}$, exist for the case where all nodes and edges in the subgraph are mapped. Again, if $L$ and $L_O$ are $\Sigma_E$ we omit the superscript.

All definitions use $\sim$ to express first that each $L$-child of the first node must be mapped to a corresponding child of the second node such that the two children are deep equal. Second, the definition ensures that the order of edges, where they are ordered in the first place and covered in $L_O$, is preserved by the mapping. The definition takes care that the order specification of the data takes precedence over $L_O$, i.e., membership in

$$\mathsf{nds}^L : N \to \wp(\mathbb{N}) = \left\{(n, N_n) \;\middle|\; n \in N \wedge N_n = \{n' \in N : \exists \lambda \in L : \lambda(n, n')\} \cup \{n\}\right\}$$

$$\mathsf{nds}^L_* : N \to \wp(\mathbb{N}) = \left\{(n, N_n) \;\middle|\; n \in N \wedge N_n = \{n' \in N : \exists \lambda \in L : \lambda_+(n, n')\} \cup \{n\}\right\}$$

$$\mathsf{eds}^L : N \to \wp(\mathbb{N}) = \left\{(n, E_n) \;\middle|\; n \in N \wedge E_n = \{(n, i) \in N \times \mathbb{N} : \exists n' \in \mathsf{nds} : (n, n', i) \in E\}\right\}$$

$$\mathsf{eds}^L_* : N \to \wp(\mathbb{N}) = \left\{(n, E_n) \;\middle|\; n \in N \wedge E_n = \{(n', i) \in N \times \mathbb{N} : \exists n', n'' \in \mathsf{nds}^* : (n', n'', i) \in E\}\right\}$$

$$\begin{aligned}
\sim^{L,L_O} = \Big\{(n, m, f, g, \triangleq) \;\Big|\; & \forall c_n \in N, i \in \mathbb{N} : e = (n, c_n, i) \in E \wedge \mathfrak{L}(e) \in L \implies \\
& \exists i' \in \mathbb{N} : g(n, i) = (m, i') \wedge e' = (m, f(c_n), i') \in E \wedge \cong (e, e') \wedge \triangleq^{L,L_O} (c_n, f(c_n)) \\
& \wedge (\lambda \in L_O \cap \mathfrak{D}(n) \implies \forall \lambda' \in L_O \cap \mathfrak{D}(n), k \in \mathbb{N} : \\
& \quad e'' = (n, f(c_n), k) \in E \wedge \mathfrak{L}(e'') = \lambda' \wedge k \leq i \implies \exists k' : g(n, k) = (m, k') \wedge k' \leq i'\Big\}
\end{aligned}$$

$$\begin{aligned}
\triangleq^{L,L_O} = \Big\{(n, m) \in N^2 \;\Big|\; & =_{\mathsf{label}} (n, m) \wedge \mathfrak{D}(n) = \mathfrak{D}(m) \wedge \Big(\exists f : \mathsf{nds}^L(n) \to \mathsf{nds}^L(m), f' : \mathsf{nds}^L(m) \to \mathsf{nds}^L(n), \\
& g : \mathsf{eds}^L(n) \to \mathsf{eds}^L(m), g' : \mathsf{eds}^L(m) \to \mathsf{eds}^L(n) : \wedge \sim^{L,L_O} (n, m, f, g, \triangleq) \wedge \sim^{L,L_O} (m, n, f', g', \triangleq)\Big)\Big\}
\end{aligned}$$

$$\begin{aligned}
\triangleq^{L,L_O}_{bij} = \Big\{(n, m) \in N^2 \;\Big|\; & =_{\mathsf{label}} (n, m) \wedge \mathfrak{D}(n) = \mathfrak{D}(m) \wedge \Big(\exists f : \mathsf{nds}^L(n) \to \mathsf{nds}^L(m), g : \mathsf{eds}^L(n) \to \mathsf{eds}^L(m) : \\
& f, g \text{ bijective} \wedge \sim^{L,L_O} (n, m, f, g, \triangleq_{bij}) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \triangleq_{bij})\Big)\Big\}
\end{aligned}$$

$$\begin{aligned}
\triangleq^{L,L_O}_{*} = \Big\{(n, m) \in N^2 \;\Big|\; & \cong (n, m) \wedge \mathfrak{D}(n) = \mathfrak{D}(m) \wedge \Big(\exists f : \mathsf{nds}^L_*(n) \to \mathsf{nds}^L_*(m), g : \mathsf{eds}^L_*(n) \to \mathsf{eds}^L_*(m) : \\
& \wedge \sim^{L,L_O} (n, m, f, g, \triangleq_*) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \triangleq_*)\Big)\Big\}
\end{aligned}$$

$$\begin{aligned}
\triangleq^{L,L_O}_{*,bij} = \Big\{(n, m) \in N^2 \;\Big|\; & \cong (n, m) \wedge \mathfrak{D}(n) = \mathfrak{D}(m) \wedge \Big(\exists f : \mathsf{nds}^L_*(n) \to \mathsf{nds}^L_*(m), g : \mathsf{eds}^L_*(n) \to \mathsf{eds}^L_*(m) : \\
& f, g \text{ bijective} \wedge \sim^{L,L_O} (n, m, f, g, \triangleq_{*,bij}) \wedge \sim^{L,L_O} (m, n, f^{-1}, g^{-1}, \triangleq_{*,bij})\Big)\Big\}
\end{aligned}$$

Figure 29. DEEP EQUAL ($L_O \subset L \subset \Sigma_E$)

$L_O$ preserves order of respective edges only if the edges are considered ordered in the data.

### 5.6.7 INVERSE AND COMPLEMENT

We conclude the set of relations on data graphs by inverse and complement relations for each of the previously defined ones.

COMPLEMENT RELATIONS. For each of the basic binary relations $R$ on nodes and/or edges (but not on integers), we introduce the *complement relation*

$$\overline{R} = \begin{cases} (N \cup E)^2 \setminus R & \text{if } R \subset N \cup E \text{ label or identity equivalence relation} \\ N^2 \setminus R & \text{if } R \subset N^2 \text{ structural equiv., path, or node order rel.} \\ E^2 \setminus R & \text{if } R \subset E^2 \text{ edge order} \\ (N \times E) \setminus R & \text{if } R \subset N \times E \text{ source or sink relation} \\ N \setminus R & \text{if } R \subset N \text{ root relation} \\ N \cup E \setminus R & \text{if } R \subset N \cup E \text{ and not } R \subset N \text{ label relation} \end{cases}$$

In CLOG and the CLOCAG algebra these relations are not strictly needed and can be simulated by a complement operation over the original relation.

INVERSE RELATIONS. For each of the basic binary relations $R$, we also introduce the *inverse relation* $R^{-1} = \{(x, y) \in (N \cup E)^2 : R(y, x)\}$. Though these relations do not add to the expressiveness of conjunctive queries as defined here (cf., e.g., [170]) they can be exploited to rewrite certain classes of graph queries to tree queries as described in [169]. Consider, e.g., the graph query $Q(m) \rightarrow \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_3, v_2) \wedge \text{CHILD}(v_2, v_4)$. This can be rewritten to $Q'(m) \rightarrow \text{CHILD}(v_1, v_2) \wedge \text{CHILD}^{-1}(v_2, v_3) \wedge \text{CHILD}(v_2, v_4)$, where the relation between $v_3$ and $v_2$ is inverted making the query tree-shaped.

### 5.6.8 EXAMPLE RELATIONS

We conclude the discussion of relations on data graphs by looking back to the data from Figure 23. For that data, Table 2 gives some of the relations derived from the data graph in accordance to the above definitions.

$\mathsf{Lab}^{\text{ATTR}} = \{(d_4, 3, d_5), (d_4, 2, d_{10}), (d_{11}, 2, d_{14}), (d_1, 5, d_{20})\}$

$\mathsf{Lab}^{\text{PAPER}} = \{d_4, d_{11}\}$ $\quad$ $\mathsf{Lab}^{\text{'CICERO'}} = \{d_8, d_{17}\}$

$\cong = \{(d_7, d_{12}), (d_4, d_{11}), (d_8, d_{17}), (d_{16}, d_{18}), \ldots\}$

$\doteq = \{(d_1, d_1), (d_2, d_2), \ldots, ((d_1, 1, d_2), (d_1, 1, d_2)), \ldots\}$

$\triangleq = \{(d_8, d_{17})\}$ $\quad$ $\triangleq^{\{\text{CHILD,ATTR}\}} = \{(d_8, d_{17}), (d_4, d_{11}), (d_{16}, d_{18})\}$

$\circ\!\!\rightarrow = \{(d_1, (d_1, 1, d_2)), (d_1, (d_1, 2, d_4)), \ldots\}$ $\quad$ $\rightarrow\!\!\circ = \{(d_2, (d_1, 1, d_2)), (d_4, (d_1, 2, d_4)), \ldots\}$

$\text{CHILD} = \{(d_1, d_2), (d_1, d_4), (d_1, d_{11}), (d_1, d_{15}), (d_4, d_7), \ldots\}$

$\text{CHILD}_* = \{(d_1, d_2), (d_1, d_4), (d_1, d_7), (d_1, d_{11}), (d_1, d_{12}), \ldots\}$

$\text{path}_*^{\text{CHILD,COMMENT,VALUE}} = \{(d_1, d_2), (d_1, d_3), (d_1, d_4), (d_1, d_7), (d_1, d_8), (d_1, d_9), (d_1, d_{11}), \ldots\}$

$@ = \{(d_{11}, 3), (d_{11}, 1), ((d_1, 3, d_{11}), 3), ((d_{10}, 1, d_{11}), 1), \ldots\}$

$\ll = \{(d_2, d_4), (d_4, d_{11}), (d_{11}, d_{15}), \ldots\}$ $\quad$ $\ll_+ = \{(d_2, d_4), (d_2, d_{11}), (d_2, d_{15}), \ldots\}$

$\blacktriangleleft = \{(d_3, d_4), \ldots, (d_{17}, d_{18}), \ldots\}$ $\quad$ $\ll_+ = \{(d_3, d_4), (d_3, d_5), (d_3, d_6), \ldots\}$

Table 2. (Partial) instances for data graph relations on Figure 23

## 5.7 CONCLUSION

The data model, an ordered, semi-structure data graph with *node and edge* labels, for CIQLOG and CIQCAG is an abstraction of data models for common Web query languages. As such, it provides a rich set of relations on data graphs, discussed in Section 5.6, for CIQLOG enabling CIQLOG to be the target of for the translate of large fragments of Xcerpt, XQuery, and SPARQL, see Part III. The data model also proves to be sufficiently expressive to capture both XML and RDF data, as well as Xcerpt data terms (which have, as XML, many different types of data items structured rather freely into arbitrary graphs, as in RDF). Before turning to the translation from XQuery, Xcerpt, and SPARQL, we outline CIQLOG, our formal query language on query data graphs in the following chapter.

# 6 QUERIES—CIQLOG: DATALOG¬ WITH COMPLEX RULE HEADS



## 6.1 INTRODUCTION

As formal foundation for Web queries, we introduce CIQLog, a rule-based query language tailored to semi-structured queries. CIQLog is a slightly modified variant of $\text{datalog}^{\neg}_{new}$, i.e., *datalog extended with negation and value invention*, which is most prominently represented by ILOG [132]. We only consider binary relations, but extend $\text{datalog}^{\neg}_{new}$ with a partial order on edges (in the spirit of IDLOG [191]) and some (syntactic) conveniences such as conjunction in rule heads and disjunction in rule bodies. For most of the translations in Part III and the translation to CIQCAG, we focus on the weakly recursive fragment of CIQLog, denoted as $\text{CIQLog}^{\text{wR}}$. Following [132], a CIQLog program is weakly recursive, if there is no recursion *through value invention*.

Note, that CIQLog's value invention is, as that of ILOG [132] or RDF query languages such as RDFLog [63], based on the essential observation that the actual invented value (in our case node or edge) carries no information

whatsoever. Only its membership in relations is material. The same applies for edge positions: We do not care how they are represented (though we usually choose integers) as long as they carry the correct order relations. In particular, there is no requirement that the edge positions are consecutive and thus no need for a successor relation. This allows for more flexibility in the realization of edge positions in the algebra.

## 6.2 CIQLOG SYNTAX

In the following, we briefly summarize the syntax and semantics of CIQLog: A CIQLog rule $R$ consists of a query *head* and a query *body*. The query body is a (quantifier-free) formula over binary and unary atoms. Each atom is a relation over query variables from the underlying relational structure $D$. The domain of the query variables is the domain of $D$. The query head is mostly a conjunction of atoms over answer variables and outlined in detail below in Section 6.2.1. All answer variables must occur also in the body of the query. All other variables in the query body are existentially quantified [2].

Each variable must be either a *node, edge, edge position, or label variable*, i.e., the domain of each variable is either node, edge, edge position, or label. If a rule contains a variable that occurs at the position of two or more attributes with different domains, this rule is *invalid*, otherwise it is *valid*.

*Safety: range restriction*

Answer variables are variables that occur in the head outside of the condition of a conditional expression. The usual safety restrictions [2] for datalog apply to ensure that all rules are *range-restricted*: For each negation, all answer variables must occur also in a positive expression in the rule body. For each disjunction, all nested expressions have the same answer

| | | |
|---|---|---|
| ⟨*program*⟩ | ::= | ⟨*rule*⟩* |
| ⟨*rule*⟩ | ::= | ⟨*head*⟩ '⟵' ⟨*expression*⟩ |
| ⟨*expression*⟩ | ::= | ⟨*atom*⟩ \| ⟨*negation*⟩ \| ⟨*conjunction*⟩ \| ⟨*disjunction*⟩ |
| ⟨*atom*⟩ | ::= | ⟨*relation*⟩ '(' ⟨*variable*⟩ (',' ⟨*variable*⟩ )? ')' |
| ⟨*negation*⟩ | ::= | ¬ '(' ⟨*expression*⟩ ')' |
| ⟨*conjunction*⟩ | ::= | '('⟨*expression*⟩ '∧'⟨*expression*⟩ ')' |
| ⟨*disjunction*⟩ | ::= | '('⟨*expression*⟩ '∨'⟨*expression*⟩ ')' |

Table 3. CIQLog syntax (without heads)

| | | |
|---|---|---|
| ⟨*head*⟩ | ::= | ⟨*hexpression*⟩ |
| ⟨*hexpression*⟩ | ::= | ⟨*hatom*⟩ \| ⟨*hconjunction*⟩ \| ⟨*conditional*⟩ \| '()' |
| ⟨*hatom*⟩ | ::= | ⟨*hrelation*⟩ '(' ⟨*hterm*⟩ (',' ⟨*hterm*⟩ )? ')' |
| ⟨*hterm*⟩ | ::= | ⟨*variable*⟩ \| ⟨*invention*⟩ \| ⟨*order*⟩ \| ⟨*aggregation*⟩ |
| ⟨*invention*⟩ | ::= | $\mathsf{new}^{\langle identifier\rangle}_{\langle equivalence\text{-}rel\rangle}$('(' ⟨*variable*⟩+ ')')? |
| ⟨*aggregation*⟩ | ::= | (**sum**\| **count**\| **max**\| **min**\| **avg**) '(' ⟨*variable*⟩ ')' |
| ⟨*order*⟩ | ::= | $\mathsf{order}_{\langle order\text{-}rel\rangle}$'(' ⟨*order*⟩?, ⟨*offset*⟩, ⟨*variable*⟩* ')' |
| ⟨*hconjunction*⟩ | ::= | '(' ⟨*hexpression*⟩ '∧' ⟨*hexpression*⟩ ')' |
| ⟨*conditional*⟩ | ::= | '**if**' ⟨*condition*⟩ '**then**' ⟨*hexpression*⟩ '**else**' ⟨*hexpression*⟩ |
| ⟨*condition*⟩ | ::= | ⟨*variable*⟩ ('=' \| '≠') '**nil**' |
| ⟨*offset*⟩ | ::= | ⟨*integer*⟩ |

Table 4. Heads of CIQLog rules

variables. Finally, each answer variable must also occur in the body.

### 6.2.1 COMPLEX HEADS

CIQLog differs notably from standard datalog in the shape of rule heads, as specified in Table 4:

**(1)** VALUE AND ORDER INVENTION: CIQLog allows terms over answer variables for rule and order invention where datalog$^{\neg}$ allows only constants and variables. Value invention is the same as in datalog$^{\neg}_{new}$ though we use a term notation as in datalog$^{\neg}_{obj}$: A value invention term is a term over the invention variables with function symbol new parametrized by an equivalence relation and an identifier. The identifier allows multiple value invention statements in the same head each returning a distinct set of new values. The equivalence relation is used to determine when two binding tuples for the invention variables are considered equivalent (and thus yield the same new value). Otherwise, a value invention term is interpreted as any other function term on nodes or edges and maps to either a node, edge, or edge position (but not a label as new labels can not be invented). As for variables in the body, value invention terms may occur either at the position of node-, edge-, *or* edge position-valued attributes. If the same value invention term occurs at the position of two or more attributes with

different domains, that rule is *invalid*. In the following, we only consider *valid* CIQLOG rules.

**Order invention terms** are similar but map to integer values (rather than nodes, edges, or edge positions). Furthermore, they are parametrized by order relations rather than equivalence relations. Order terms are used to control position among elements and are nested to allow the expression of complex sequences resulting from nested grouping expressions in languages such as XQuery or Xcerpt.

To illustrate the use of order terms, first consider the following example: Given binding tuples over the variables $x_a$, $x_b$ and $x_c$, we want to construct a new node for each distinct value of $x_a$. For this task we can use value invention terms for creating a new node, e.g., $\mathsf{new}^{\mathsf{id}_1}_{\doteq}(x_a)$. This term provides a new node for each distinct value of $x_a$ (w.r.t. $\doteq$).

As children of this node, a list of b's and c's is asked for, but in such a way that for each distinct value of $x_b$ a new b is created and for each distinct value of $x_c$ a new c. These new nodes should be children of the node created above for the corresponding value of $x_a$. Furthermore, a c created for a value of $x_c$ should be placed after the b created for the corresponding value of $x_b$ with only c's inbetween. To achieve this we use the following order terms: $t_1 = \mathsf{order}_{<_N}(\top, \mathsf{o}, x_b)$ and $t_2 = \mathsf{order}_{<_N}(\mathsf{order}_{<_N}(\top, 1, x_b), \mathsf{o}, x_c)$. E.g., for the binding tuples $(x_a = 1_a, x_b = 1_b, x_c = 1_c)$ and $(x_a = 1_a, x_b = 2_b, x_c = 1_c)$ we obtain $t_{1,1} = \mathsf{order}_{<_N}(\top, \mathsf{o}, 1_b)$ und $t_{2,1} = \mathsf{order}_{<_N}(\mathsf{order}_{<_N}(\top, 1, 1_b), \mathsf{o}, 1_c)$ for the first tuple and $t_{1,2} = \mathsf{order}_{<_N}(\top, \mathsf{o}, 2_b)$ und $t_{2,2} = \mathsf{order}_{<_N}(\mathsf{order}_{<_N}(\top, 1, 2_b), \mathsf{o}, 1_c)$ for the second tuple. We say for two order terms $t_1 = \mathsf{order}_{<_N}(g, o, \vec{x})$ and $t_2 = \mathsf{order}_{<_N}(g', o', \vec{x}')$ that $t_1 < t_2$ in two cases: **(a)** If $g = g'$ and $\vec{x} <_N \vec{x}'$ or $\vec{x} =_N \vec{x}'$ and $o < o'$. This case covers the comparison of two order terms at the same "level". In the above example, the order terms $t_{1,1}$ and $t_{1,2}$, e.g., are at the same level and thus can be compared directly. Direct comparison is performed by looking at the binding vectors $\vec{x}$ and $\vec{x}'$ and, if those are the same, using the offsets $o$ and $o'$ as "tie brakers". **(b)** If $g \neq g'$, we look also at the nested order terms: Intuitively, we look for the outermost two order terms contained in $t_1$ and $t_2$ that are comparable using <, i.e, that have the same order term as first component $g$. Formally, $t_1 < t_2$ if there are order terms $t_1'$ and $t_2'$ with $t_1' = t_1$ or $t_1'$ contained in $t_1$ and $t_2'$ respectively such that (i) $t_1' < t_2'$ and (ii) there are no two order terms $t_1''$ in $t_1$ and $t_2''$ in $t_2$ such that $t_1'' > t_2''$ and $t_1'$ contained in $t_1''$ and $t_2'$ contained in $t_2''$.

In the above example, $t_{1,1} < t_{2,1}$ as $t_{2,1}$ contains the order term $t'_{2,1} = \mathsf{order}_{<_N}(\top, 1, 1_b)$ such that $t'_{2,1} > t_{1,1}$ (since they have the same first component and binding vectors, but the offset of $t_{2,1}$ is larger than the offset of $t_{1,1}$). $t_{2,1} < t_{1,2}$ as $t'_{2,1} < t_{1,1}$ ($1_b <_N 2_b$). $t_{2,1} < t_{2,2}$ as $t'_{2,1} < t'_{2,2}$ with $t'_{2,2}$ the first component of $t_{2,2}$. The reason order terms are chosen this way is to allow flexible translation of grouping expressions from languages such as

XQuery and Xcerpt where in the same context grouping expressions for entirely different variables may occur, e.g., in

```
for $a in //a return
  <a>
    for $b in $a//b return
    (<b/>, for $c in $b//c return <c/>)
  </a>
```

where in the a we have a sequence of b and c nodes such that each b node is followed by as many c nodes as there are c descendants of the corresponding //a//b element (in the input document). The above order terms can then be employed to translate such a query, as shown in detail in Chapter 8.

We use $\top$ to denote the "empty" order term that is neither smaller nor larger than any other term (but used to complete top-level order terms) and equal only to itself. Order terms contain $\top$ or other order terms as order terms for their first component. The nesting is only on the first argument and thus linear. As usual, $\vec{x}$ and $\vec{y}$ may be empty. The empty tuple is equal only to itself and stands in $<_N$ relation to no other tuple. Order invention terms may only occur in place of edge position-valued attributes. $<_N$ is an order relation on (named) tuples of nodes or edges. A typical example of $<_N$ is the component-wise lexical order $<_{lex}$ on the label of those nodes or edges: E.g., $\langle a : x_1, b : x_2\rangle <_{lex} \langle a : y_1, b : y_2\rangle$ if $\mathfrak{L}(x_1)$ is in lexical order before $\mathfrak{L}(y_1)$ or both labels are the same and $\mathfrak{L}(x_1)$ is in lexical order before $\mathfrak{L}(y_1)$. For further examples of order terms see Chapters 7 and 8.

**(2)** CONDITIONAL CONSTRUCTION: For convenience, we allow conditional construction in the head: some part of the head depends on a condition on an answer variable (viz., that variable being **nil** or not). Conditional construction $h \wedge$ **if** $X$ = **nil then** $hc_1$ **else** $hc_2 \longleftarrow b$, can be rewritten to rules without conditional constructions as follows:

$$h \wedge hc_1 \longleftarrow b \wedge \mathrm{X} = \textbf{nil}$$
$$h \wedge hc_2 \longleftarrow b \wedge \mathrm{X} \neq \textbf{nil}$$

**(3)** AGGREGATION: As ILOG, CIQLOG extends rule heads with aggregation on integers in the spirit of [147]. Variables occurring in aggregation functions may not occur outside of aggregation functions. We denote with $\mathsf{aggVars}(R)$ the aggregation variables of a rule $R$, with $\mathsf{stdVars}(R) = \mathsf{Vars}(R) \setminus \mathsf{aggVars}(R)$ all non-aggregation variables. Aggregation has no further effect on expressiveness and complexity in presence of value invention.

Adapting the notation of [132], we call an *invention atom* an expression containing either new or order terms. The relation name of that atom is

called an *invention relation name*. A rule is a *non-invention rule*, if it contains no invention atom in the head, otherwise it is an *invention rule*. A *weakly recursive* CIQLOG program (or $\text{CIQLOG}^{\text{WR}}$ program) is a CIQLOG program where no invention rule depends (directly or indirectly) on another invention rule. We say that a program $P$ *has cascading value invention*, if there are two invention rules $R, R'$ such that $R$ depends on $R'$.

Only the first extension has an effect on the expressive power of CIQLOG. It makes CIQLOG essentially an ILOG [132] variant extended with (partial) order on edges (though it has no successor relation in contrast to IDLOG [191]).

For value invention terms, we mostly omit the equivalence relation in the following assuming identity $\doteq$.

## 6.3 CIQLOG SEMANTICS

We characterize the semantics of CIQLOG in three ways in the following:

**(1)** The intuitive semantics of a CIQLOG program is that of a logic program with aggregates [123] where we replace all occurrences of new and order in the result by unique new nodes, edges, or edge positions. The last operation is similar to un-Skolemization [82, 63], see Chapter 2.

**(2)** Together with the observation, that new and order are thus nothing else but Skolem terms with implicit relations on the results of order that can, as well, be expressed by additional CIQLOG rules, we notice that the semantics of CIQLOG can be defined by reduction of ILOG [132] whose semantics is based on Skolem terms and logic programming with aggregates.

**(3)** Finally, we give an algebraic semantics based on fixpoint and relational algebra operations that is useful both for the translations in Part III and for the equivalence to the CIQCAG algebra.

In the following, we assume that disjunction in the body and conditional construction in the head is removed as outlined above.

**Definition 6.1** (Logic-based Semantics of CIQLOG)**.** Let $P$ be a range-restricted, valid CIQLOG program. Then let $S$ be the semantic of $P$ considered as a logic program (with aggregates). If $S$ is infinite, the semantics of $P$ is undefined. Otherwise, we replace each value invention terms in $S$ with a new node, edge, or edge position (depending on the domain of the attribute it occurs in; recall that if $P$ is valid, it occurs in only one of these three types of

attributes). Order invention terms are replaced with edge positions such that the order constraints between order invention terms are preserved[1].

### 6.3.1 EXPRESSIVENESS AND COMPLEXITY

This characterization gives an intuitive and easy semantics for CIQLOG. To judge expressiveness, complexity, and completeness properties of CIQLOG the second, equivalent, characterization of the semantic of CIQLOG programs by means of ILOG is more helpful. The following theorem establishes that CIQLOG is essentially a variant of ILOG:

**Theorem 6.1.** CIQLOG *has the same expressiveness, complexity, and completeness properties as ILOG [132].*

*Proof.* Each ILOG program containing only binary relations is a CIQLOG program if we replace each invention symbol with a new (with new identifier) over all non-invention variables of the invention atom. For ILOG programs with $n$-ary relations we construct a binary decomposition of the $n$-ary relations as in RDFLog [63].

On the other hand, each CIQLOG without order terms can be transformed into an equivalent ILOG program in the following way:

**(1)** For each invention term $t$, introduce a new creation rule containing a new predicate over the invention variables of $t$ and one invention symbol. In each rule using $t$, add an atom querying that rule in the body and replace $t$ with the variable bound to the attribute at the position of the invention symbol. The resulting program is an ILOG program (contains only datalog rules and ILOG invention rules). Its semantic is the same as that of the CIQLOG program since invention symbols are replaced by Skolem terms in the semantics of ILOG.

**(2)** Each order term can be transformed analogously, but introduce cascading value invention.

**(3)** Aggregates are also allowed in ILOG.

The result of the ILOG program is up to an isomorphism between new OIDs and node, edge, and edge positions equivalent to the result of the CIQLOG program. □

**Corollary 6.1.** *From the reduction to ILOG, it follows that*

**(1)** CIQLOG *expresses all computable queries modulo copy removal, cf. [132]. Two answers are equivalent up to "copy removal" if they differ only in*

1 This can be achieved by determining some partial order on the order invention terms and assigning edge positions (integers) in accordance to that partial order.

*invented values and those invented values are structurally equivalent (according to a given form of structural equivalence as discussed in Section 5.6.6, here we consider isomorphism). In other words, there may be additional copies of invented values as long as they share the same properties and relations. This issue is closely related to the issue of lean vs. non-lean RDF graphs as answers in languages such as SPARQL or RDFLog [63].*

**(2)** CIQLOG *is (list) constructive complete (in the sense of [64]), cf. [65]. Essentially, (list) constructive queries is designed to capture precisely the queries expressible in languages such as* CIQLOG *or ILOG. It coincides with the class of queries where the new domain elements in the output can be viewed as hereditarily finite lists constructed over the domain elements of the input. Hereditary finite lists are lists constructed over a given set U of "ur-elements" from the input domain such that each element of the list is either from U or a hereditary finite list over U.*

**(3)** CIQLOG *is* not *determinate complete (in the sense of [2]), cf. [64]. Thus, there are determinate queries that specify the specific shape for invented values (and thus are no longer domain-preserving) in the answer that can not be expressed in* CIQLOG.

**(4)** *already (negation) stratified* CIQLOG *expresses all computable queries modulo copy removal, cf. [65].*

In particular, if we limit recursion to non-invention rules, we can "postpone" value invention to the very end of query evaluation:

**Corollary 6.2.** *The weakly recursive* CIQLOG *fragment* $\text{CIQLOG}^{wR}$ *has the same data and program complexity as datalog¬: P-, resp., NEXPTIME-complete.*

If recursion is prohibited entirely, value invention again has no effect on complexity:

**Corollary 6.3.** *Non-recursive* CIQLOG*, i.e.,* CIQLOG *where recursion is not allowed for any set of rules, has the same data and program complexity as non-recursive datalog¬: in* $AC_0$*, resp., PSPACE-complete.*

### 6.3.2 DEEP AND SHALLOW COPIES

In presence of value invention, the creation of *shallow* and *deep copies* of a data item are often considered essential facilities. For CIQLOG we consider shallow and deep copy for nodes only (since edges and edge positions have no "structure"). The shallow copy of a node is a *new* node with the same label, if any, and the same children as the original node. The deep copy

of a node $n$ is a *new* node $n'$ such that $n$ and $n'$ have the same label and for each out-going edge of $n$ to a child $c_n$ there is an outgoing edge for $n'$ with the same label and same edge position to a child $c_{n'}$ such that $c_n$ and $c_{n'}$ are themselves deep copies.

We add shallow and deep clone relations to CIQLOG heads, denoted as deep-copy(X,Y) and shallow-copy(X,Y) where $Y$ is a new node and $X$ is the original node. In full CIQLOG, shallow and deep copy can be implemented as the following rules:

```
ℒ(Y,L)
⟵ deep-copy(X,Y) ∧ ℒ(X,L) ∧ ¬○→(X, E).
ℒ(Y,L) ∧ ○→(Y,new₁(X,Z)) ∧ →○(new₂(X,Z),new₁(X,Z)) ∧
    pos(new₁(X,Z),EPos) ∧ deep-copy(Z,new₂(X,Z))
⟵ deep-copy(X, Y) ∧ ℒ(X, L) ∧ ○→(X, E) ∧ →○(Z, E) ∧ pos(E,
    EPos).
```

```
ℒ(Y,L)
⟵ shallow-copy(X,Y) ∧ ℒ(X,L) ∧ ¬○→(X,E).
ℒ(Y,L) ∧ ○→(Y,new₁(X,Z)) ∧ →○(Z,new₁(X,Z)) ∧ pos(new₁(X,Z),EPos)
⟵ shallow-copy(X,Y) ∧ ℒ(X,L) ∧ ○→(X,E) ∧ →○(Z,E) ∧ pos(E,EPos).
```

However, the resulting program has necessarily cascading value invention. Furthermore, the rule-based realization is, in general, less efficient than a specialized operator. To provide deep- and shallow-copy also to CIQLOG$^{\text{wR}}$, we define them as specialized operators with the above semantics. Note, that both deep- and shallow-copy are linear time, constant additional space operations and run in $\mathcal{O}(|N|+|E|)$. They are considered value invention operators and, thus, may in CIQLOG$^{\text{wR}}$ not occur in recursive rules.

### 6.3.3 ALGEBRAIC SEMANTICS

A third characterization of the CIQLOG semantics is given by a translation of CIQLOG rules to relational algebra expressions with value invention. Combined with a fixpoint operator, the resulting expressions yield a semantics of CIQLOG. The target language is roughly **while**$_{new}$ of [2], but uses invention terms instead of a dedicated invention relation. The advantage of this characterization is that it yields a very compact semantics closely based on relation algebra expressions, in particular, for CIQLOG$^{\text{wR}}$ and non-recursive CIQLOG.

In the following, we denote, for any sub-formula $q$ of a rule $R$, with free$(q)$ the variables in $q$ that also occur in $R$ outside of $q$. For a set of attributes $A = \{a_1, \ldots, a_n\}$ let $t \in D^A$ be an $|A|$-ary tuple $\langle a_1 : v_1, \ldots, a_n : v_n \rangle$ over the domain $D$ with $t[a_i]$ the value of $t$ for attribute $a_i$. We allow for *partial relational structures* $D$ where relation instances exists only for

$$
\begin{aligned}
&[\![\, head \leftarrow expr \,]\!] && = [\![\, head \,]\!]_h\,([\![\, expr \,]\!]_b)\\
&[\![\, rel(x_1,\ldots,x_n) \,]\!]_b && = \left\{ t \in D^{\{x_1,\ldots,x_n\}} : (t[x_1],\ldots,t[x_n]) \in [\![\, rel \,]\!]_b \right\}\\
&[\![\, \neg(expr) \,]\!]_b && = \left\{ t \in D^A : A = \mathsf{free}(expr) \wedge t \in D^A \setminus \pi_A([\![\, expr \,]\!]_b) \right\}\\
&\left[\!\!\left[\, (expr_1 \wedge expr_2) \,\right]\!\!\right]_b && = \pi_A(\left[\!\!\left[\, expr_1 \,\right]\!\!\right]_b \bowtie \left[\!\!\left[\, expr_2 \,\right]\!\!\right]_b),\ A = \mathsf{free}(expr_1 \wedge expr_2)\\
&\left[\!\!\left[\, (expr_1 \vee expr_2) \,\right]\!\!\right]_b && = \left[\!\!\left[\, \pi_A(expr_1) \,\right]\!\!\right]_b \cup \pi_A(\left[\!\!\left[\, expr_2 \,\right]\!\!\right]_b),\ A = \mathsf{free}(expr_1)
\end{aligned}
$$

---

$$
\begin{aligned}
&[\![\, expr \,]\!]_h\,(\beta) && = \mathsf{subst}(expr, \beta)\\
&\left[\!\!\left[\, (hexpr_1 \wedge hexpr_2) \,\right]\!\!\right]_h && = \left[\!\!\left[\, hexpr_1 \,\right]\!\!\right]_h \uplus \left[\!\!\left[\, hexpr_2 \,\right]\!\!\right]_h\\
&\left[\!\!\left[\, \texttt{if}\ c\ \texttt{then}\ hexpr_1\ \texttt{else}\ hexpr_2 \,\right]\!\!\right]_h && = \left[\!\!\left[\, hexpr_1 \,\right]\!\!\right]_h \text{ if } [\![\, c \,]\!]_h = \textbf{true},\ \left[\!\!\left[\, hexpr_2 \,\right]\!\!\right]_h \text{ otherwise}\\
&[\![\, () \,]\!]_h && = \varnothing\\
&[\![\, rel(t_1,\ldots,t_n) \,]\!]_h && = \{rel \mapsto \{t_1,\ldots,t_n\}\}\\
&[\![\, x = \texttt{nil} \,]\!]_h && = \textbf{true} \text{ if } x = \texttt{nil},\ \textbf{false} \text{ otherwise}\\
&[\![\, x \neq \texttt{nil} \,]\!]_h && = \textbf{true} \text{ if } x \neq \texttt{nil},\ \textbf{false} \text{ otherwise}\\
&[\![\, \mathsf{deep\text{-}copy}(x,y) \,]\!]_h && = \mathsf{deep\text{-}copy}(x,y)\\
&[\![\, \mathsf{shallow\text{-}copy}(x,y) \,]\!]_h && = \mathsf{shallow\text{-}copy}(x,y)
\end{aligned}
$$

Table 5. Algebraic CIQLOG Semantics ($D$ the domain of the relational structure)

a subset of the relations in the underlying relational schema. We denote that a set of tuples $T$ over attributes $U$ is an instance of a relation schema $RN[U]$ by $RN \mapsto T$ and consider a partial relational structure as a set of such mappings. A partial relational structure is *complete* wrt. a relational schema $S$ if it contains mappings for all relation schemas in $S$. We call $D' = D_1 \uplus D_2$ the *union of partial relational structures* such that $D' = \{RN \mapsto T_1 \cup T_2 : RN \mapsto T_1 \in D_1 \wedge RN \mapsto T_2 \in D_2\}$ where $RN \mapsto \varnothing$ if there is no mapping of $RN$ in a relational structure.

Head formulas $E$ are instantiated by substitution, denoted by $\mathsf{subst}(E, \beta)$ which returns the set of all expressions $E$ replacing variable occurrences by bindings from $\beta$:

$$
\begin{aligned}
\mathsf{subst}(E,\beta) = \biguplus_{t \in \pi_{\mathsf{stdVars}(E)}(\beta)} \Big( & [\![ E\{x/t[x] : x \in \mathsf{stdVars}(E)\}\\
& \{\textbf{sum}(y)/\textstyle\sum\{t'[y] : t' \in \beta \wedge t = \pi_{\mathsf{stdVars}(E)}(t')\} : y \in \mathsf{aggVars}(E)\}\\
& \{\textbf{max}(y)/\max\{t'[y] : t' \in \beta \wedge t = \pi_{\mathsf{stdVars}(E)}(t')\} : y \in \mathsf{aggVars}(E)\}\\
& \{\textbf{min}(y)/\min\{t'[y] : t' \in \beta \wedge t = \pi_{\mathsf{stdVars}(E)}(t')\} : y \in \mathsf{aggVars}(E)\}\\
& \{\textbf{avg}(y)/\mathrm{avg}\{t'[y] : t' \in \beta \wedge t = \pi_{\mathsf{stdVars}(E)}(t')\} : y \in \mathsf{aggVars}(E)\}\\
& \{\textbf{count}(y)/|\{t'[y] : t' \in \beta \wedge t = \pi_{\mathsf{stdVars}(E)}(t')\}| : y \in \mathsf{aggVars}(E)\} ]\!]_h \Big)
\end{aligned}
$$

For aggregation free formulas, this resumes to standard substitution, whereas aggregation expressions are substituted by their respective aggregation function over the aggregated values of the aggregation variables.

Using these definitions, Table 5 gives the algebraic semantics of a CIQLOG rule. Recall, that all rules are range-restricted. We use $[\![\ ]\!]_b$ to define the semantics of the body of a rule, $[\![\ ]\!]_h$ that of a head. The body of a rule results in a single relation over the free variables of the query. The head of a rule is evaluated once for each tuple resulting from the evaluation of the body of the query, each time replacing all the occurrences of all query variables by their bindings. A deep copy operation is replaced by a relational structure $D$ containing $y$ and a copy of all nodes reachable from $x$ as well as their relations. A shallow copy operation is replaced by a relational structure $D$ containing $y$, the same label relation on $y$ as exists on $x$, and all nodes reachable from $x$ (excluding $x$) and their relations as well as edges from $y$ to all children of $x$. The result of $[\![\ ]\!]_h$ is a *pre-instance* in the sense of [132], i.e., it still contains value and order invention terms. These are replaced by new nodes, edges, and edge positions accordingly as described above. Computing an instance from an pre-instance can be done by assembling the order invention terms in a partial order, and then replacing them according to that order. Nested order terms do not pose a challenge, as we can replace the nested terms by first ordering the depth 1 terms, then add the depth 2 (respecting the order among the depth 1 terms contained in those depth 2 terms, etc. Together with the replacement of node invention terms that computation is in $\mathcal{O}(o \cdot \log o)$ time using $\mathcal{O}(\log o)$ additional space where $o$ is the size of the pre-instance, assuming in-place sorting. Thus, the replacement of value and order invention terms does not affect the overall complexity of evaluating a single rule in CIQLOG (which is the complexity of relational algebra, i.e., in L wrt. data complexity, PSPACE wrt. query complexity).

To obtain a semantics for full CIQLOG we complement the above evaluation of a single rule (given the current set of derived facts represented as a relational structure) by a standard fixpoint operator (or iteration construct such as used in [2] for $\mathbf{while}_{new}$).

**Theorem 6.2.** *A single application of the fixpoint operator on the algebraic semantics yields an equivalent result as a single application of the fixpoint operator on the logic-based semantics.*

*Proof.* For simplicity, we assume stratified negation. Recall, from [65] that, in contrast to Datalog, the restriction to negation stratified programs does not limit the expressiveness of ILOG and thus CIQLOG. Furthermore, we limit ourselves to rules without conjunction, conditional construction, and deep or shallow copy in the head or disjunction in the body (all these

features can be rewritten to equivalent CIQLOG programs beforehand).

It is easy to verify that the algebraic semantics given in Table 5 on rules without those features represents all facts derivable from the given relational structure (representing the facts derived by the last application of the fixpoint operator) by the rule. Note, that the result is a relational structure containing a non-empty instance for exactly one relation, viz. the one of the head atom. □

This concludes our discussion of the CIQLOG semantics. Before we put CIQLOG to work in evaluating Xcerpt, XQuery, and SPARQL in Part III, we discuss in the following, briefly, the querying of data graphs as introduced in Chapter 5, as well as the fragment of CIQLOG used primarily for the translations in Part III.

## 6.4 DATA GRAPHS IN CIQLOG: EXTENSIONAL AND INTENSIONAL RELATIONS

In CIQLOG, only a small set of relations on data graphs need to be extensional, i.e., represented as sets of tuples. The remaining relations can be realized by intensional definitions as CIQLOG rules on top of this set.

The basic, extensional relations are, unsurprisingly, $\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$, pos, root, $\mathfrak{L}$, $\mathfrak{V}$ which together provide access to all information in a data graph: about an edge, we know source, sink, edge position, and label, about a node, label, whether it is a root, its incoming and outgoing edges.

The remaining relations from Chapter 5 can be realized on top of these five basic relations by a set of CIQLOG rules as shown in Figure 30. Rules for inverse and complement relations are defined as usual. Note, that $\overline{R}$ is only defined for relations $R$ on nodes and/or edges (not on integers) both of which are finite domains that can be enumerated using $\circ\!\!\rightarrow$, $\rightarrow\!\!\circ$, and root. We only show rules for one case of deep equal, the remaining cases are similar, though considerably more involved if bijective mappings are required as we need to track the actual mapping to ensure bijectivity.

In Figure 30, we give rules for arbitrary label sets $\{\lambda_1, \ldots, \lambda_n\}$ (line 1 ff.) and path lengths ($i$, $j$ in line 15–18). Both path lengths and label sets are size limited by the label alphabet, resp. the number of edges in the data. However, we usually assume a much smaller number of such relations for any particular query task. E.g., in Part III each of the discussed translation uses no at most ternary label sets and no more than two or three path relations.

*Common expressions over data graph relations*

We use some abbreviations for common expressions over data graph relations in CIQLOG heads: First, instead of $\mathsf{new}^{\mathsf{id}_1}_{\doteq}(x_1, \ldots, x_n)$ we write simply $\mathsf{id}_1(x_1, \ldots, x_n)$. Second, we use $RN(n, m, i)$ for $\circ\!\!\rightarrow(n, \mathsf{id}) \wedge \rightarrow\!\!\circ(m, \mathsf{id}) \wedge$

```
Lab^{λ1,...,λn}(X)
⟵ 𝔏(X,λ1) ∨ 𝔏(X,λ2) ∨ ... ∨ 𝔏(X,λn).

𝔒^{λ1,...,λn}(X)
⟵ 𝔒(X,λ1) ∨ 𝔒(X,λ2) ∨ ... ∨ 𝔒(X,λn).

@^S(E, count(EPos')+1)
⟵ pos(E,EPos) ∧ ○→(N,E) ∧ ○→(N,E') ∧ Lab^S(E') ∧ 𝔏(E', EL) ∧
    𝔒(N, EL) ∧ pos(E',EPos') ∧ EPos < EPos'.
@^S(N,EPos)
⟵ →○(N,E) ∧ @^S(E,EPos).

indeg^S(N,count(E)) ⟵ →○(N,E) ∧ Lab^S(E).
outdeg^S(N,count(E)) ⟵ ○→(N,E) ∧ Lab^S(E).

path^S_{i,j}(N,N')
⟵ i < j ∧ ¬(N ≐ N') ∧ ○→(N,E1) ∧ Lab^S(E1) ∧
    →○(N1,E1) ∧ path^S_{max(0,i-1),j-1}(N1,N'))
path^S_{0,j}(N,N) ⟵ 0 ≤ j.

<^S(E,E') ⟵ @^S(E,EPos) ∧ @^S(E',EPos+1).
<^S_+(E,E') ⟵ @^S(E,EPos) ∧ @^S(E',EPos') ∧ EPos < EPos'.
<^S_*(E,E') ⟵ @^S(E,EPos) ∧ @^S(E',EPos') ∧ EPos ≤ EPos'.
≪^S(N,N') ⟵ →○(N,E) ∧ →○(N',E') ∧ E <^S E'.
≪^S_+(N,N') ⟵ →○(N,E) ∧ →○(N',E') ∧ E <^S_+ E'.
≪^S_*(N,N') ⟵ →○(N,E) ∧ →○(N',E') ∧ E <^S_* E'.

◂^S_+(N,N') ⟵ path^S_*(N,M) ∧ path^S_*(N',M') ∧ M ≪^S_+ M'.
◂^S_*(N,N') ⟵ path^S_*(N,M) ∧ path^S_*(N',M') ∧ M ≪^S_* M'.
◂^S(N,N') ⟵ N ◂^S_+ N' ∧ ¬(N ◂^S_+ M ∧ M ◂^S_+ N' ∨ path^S_*(M,N')).

≅(X,X') ⟵ 𝔏(X,L) ∧ 𝔏(X',L).
≐(N,N) ⟵ ○→(N,E) ∨ →○(N,E) ∨ root(N).
≐(E,E) ⟵ ○→(N,E).
=^S_@(E,E') ⟵ @^S(E,EPos) ∧ @^S(E',EPos).

≗(N,N') ⟵ N ≐ N'.
≗(N,N')
⟵ N ≅ N' ∧ ¬(○→(N,E) ∧ →○(M,E) ∧
    ¬(○→(N',E') ∧ →○(M',E') ∧ M ≗ M')).
```

Figure 30. CIQLOG rules for intensional data graph relations

$\mathfrak{L}(\mathsf{id}, RN) \wedge \mathsf{pos}(\mathsf{id}, i)$ where $\mathsf{id}$ is a new value invention term not used in the rest of the program. Third, we use $RN(n, m)$ for $\multimap(n, \mathsf{id}) \wedge \multimap(m, \mathsf{id}) \wedge \mathfrak{L}(\mathsf{id}, RN) \wedge \mathsf{pos}(\mathsf{id}, \mathsf{id}')$ where $\mathsf{id}$ and $\mathsf{id}'$ are new value invention terms not used in the rest of the program. In the latter case, we do not care about the actual edge position and thus allow an arbitrary one. It should only be used if $(n, RN) \notin \mathfrak{O}$.

## 6.5 NON-RECURSIVE CIQLOG

Given these low numbers of actually used relations, the evaluation of CIQLog queries profits from precomputing of or providing special access operators to most or all of the derived relations discussed before. This allows us to define the final sub-language of CIQLog, non-recursive or single-rule CIQLog. A non-recursive CIQLog program consists in a single, non-recursive CIQLog rule (that may use any of the data graph relations). Thus its semantics can, given special access operators or precomputation of all data graph relations, be specified without use of a fixpoint or similar recursion construct as shown in Table 5. Yet such programs are equivalent to $\text{CIQLog}^{\text{wR}}$ programs comprised of the single rule of the original program and the rules from Figure 30 for the derived data graph relations. Note, that for non-recursive CIQLog the restriction to a single-rule has no effect on expressiveness due to disjunction in rule bodies and conditional construction in the head. Single-rule CIQLog considerably simplifies the correctness proofs in Part III as well as the transformation to CIQCAG.

We assume in the following that the head of a single-rule CIQLog program is not matched against its body. This can be achieved by adding a root node to the head of that rule with a label that is not matched by an added root node test in the body (and connections between that root node and all nodes and edges in the query). In the following, we omit these parts of a CIQLog rule when talking about non-recursive CIQLog.

Precomputing these relations is acceptable for unary or integer-valued relations such as Lab, @ or indeg. However, for node-node or edge-edge relations such as path relations, order relations or equivalence relations, the space cost may be prohibitive and thus specialized operators might be preferable.

Using the approaches outlined in the following sections, we obtain for all CIQLog relations (except deep equal) constant membership test at a space cost of only $\mathcal{O}(|D|) = \mathcal{O}(|N| + |E|)$. On tree data, this is obviously $\mathcal{O}(|N|)$.

### 6.5.1 REACHABILITY IN DATA GRAPHS

Order and path relations can be seen as variants of reachability on the underlying base relations. For tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [86] or other labeling schemes such as [200]). However, for graph data this is not so obvious. Fortunately, there has been considerable research on reachability or closure relations and their indexing in arbitrary graph data in recent years. Table 7 summarizes the most relevant approaches for our work. Theoretically, we can obtain constant time for the membership test if we store the full transitive closure matrix. However, for large graphs this is clearly infeasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test.

The first class are based on the idea of a 2-hop cover [76]: Instead of storing a full transitive closure, we allow that reachable nodes are reached via at most one other node (i.e., in two "hops"). More precisely, each node $n$ is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach $n$, $out(n)$ a set of nodes that are reachable from $n$. Both sets are assigned in such a way, that a node $m$ is reachable from $n$ iff $out(n) \cup in(m) \neq \varnothing$. Unfortunately, computing the optimal 2-hop cover is NP-hard and even advanced approximation algorithms [189] have still rather high complexity.

A different approach [5, 70, 199, 195] is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [199] where a full transitive closure over the non-tree edges is build. GRIPP [195] and SSPI [70] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

For a specialized reachability test operator in CIQLOG we can choose any of the approaches. For the following, we assume constant time membership, since that is easily achieved on trees and feasible with approaches such as Dual Labeling even for graphs.

### 6.5.2 EQUIVALENCE IN DATA GRAPHS

Most of the equivalence relations can, again, be easily computed from unary base relations such as ♀. However, this is not the case for deep equality. Indeed, the deep equality comes at a considerable higher cost

| approach | characteristics | query time | index time | index size |
|---|---|---|---|---|
| **Shortest path** [174] | no index | $\mathcal{O}(n+e)$ | – | – |
| **Transitive closure** | full reachability matrix | $\mathcal{O}(1)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^2)$ |
| **2-Hop** [76] | 2-hop cover[a] | $\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n \cdot \sqrt{e})$ |
| **HOPI** [189] | 2-hop cover, improved approximation algorithm | $\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n \cdot \sqrt{e})$ |
| **Graph labeling** [5] | interval-based tree labeling and propagation of intervals of non-tree descendants. | $\mathcal{O}(n)^b$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^2)^c$ |
| **SSPI** [70] | interval-based tree labeling and recursive traversal of non-tree edges | $\mathcal{O}(e-n)$ | $\mathcal{O}(n+e)$ | $\mathcal{O}(n+e)$ |
| **Dual labeling** [199] | interval-based tree labeling and transitive closure over non-tree edges | $\mathcal{O}(1)^d$ | $\mathcal{O}(n+e+e_g^3)$ | $\mathcal{O}(n+e_g^2)$ |
| **GRIPP** [195] | interval-based tree labeling plus additional interval labels for edges with incoming non-tree edges | $\mathcal{O}(e-n)$ | $\mathcal{O}(n+e)$ | $\mathcal{O}(n+e)$ |

***a*** Index time for approximation algorithm in [76].

***b*** More precisely, the number of intervals per node. E.g., in a bipartite graph this can be up to $n$, but in most (sparse) graphs this is likely considerably lower than $n$.

***c*** More precisely, the total number of interval labels.

***d*** [199] introduces also a variant of dual labeling with $\mathcal{O}(\log e_g)$ query time using a, in practical cases, considerably smaller index. However, worst case index size remains unchanged.

Table 7. Cost of Membership Test for Closure Relations. $n$, $e$: number of nodes, edges in the data, $e_g$: number of non-tree edges, i.e., if $T(D)$ is a spanning tree for $D$ with edges $E_{T(D)}$, then $e_g = |E_D \setminus E_{T(D)}|$.

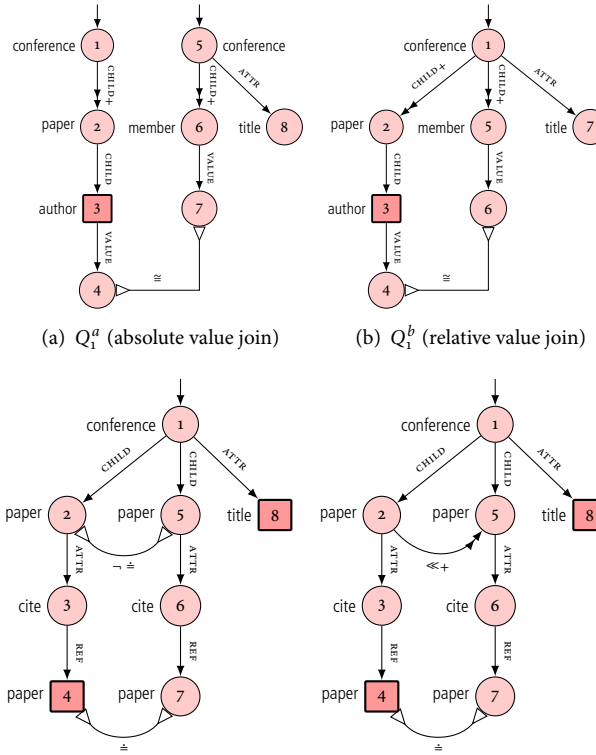
than, e.g., label equality, in particular if the data is unordered. First, if the data is fully ordered and $L_O = L$ then deep equal is in $O(j(|E|))$ where $E$ are the edges of the data graph. The same applies if either edge or node labels are keys, i.e., partition the child nodes of each node in the data in singleton sets. A straightforward parallel traversal of the two subgraphs suffices to test equivalence. In case of XML data, e.g., the data is ordered except for attributes. Attribute *labels*, however, are keys and thus do not need to be ordered for linear time complexity.

If the data is tree shaped but unordered, deep equal reduces to general tree isomorphism which can still be solved in linear time due to [126]. Moreover, for composition-free languages such as non-recursive $\text{C}^{\text{I}}\text{QLog}^{\text{wR}}$ or single-rule $\text{C}^{\text{I}}\text{QLog}$ where invented nodes are never queried and thus node invention can be seen as a post-processing step, deep equal does not add to the expressiveness of the query language if the data is tree shaped (cf. [144] on composition-free XQuery).

If the deep equality is not injective, but the data graph-shaped the complexity becomes polynomial as the problem essentially reduces to bisimulation [88]. However, on unordered *graphs* testing injective deep join of two nodes subsumes to graph isomorphism and thus is commonly believed to exhibit no polynomial algorithm (regardless of whether we consider the full subgraph or only the children or adjacent nodes for injectivity). However, even for this general case there exist reasonably fast algorithms, e.g., [158, 96].

Thus in presence of deep equal we either have to accept that membership tests for deep equal induce considerable, in most cases exponential additional cost, or we have to reconcile to precomputation of deep equal (which can be done once at a cost of $\mathcal{O}(|N|^2 \cdot |E|^{|E|})$ time). Obviously this relation can be precomputed in $O(n^2 \times n)$ for tree data. Using sub*tree* isomorphism algorithms, we can further reduce the time complexity by a linear factor. Whether there is a better algorithm for precomputing this relation in the general case, remains an open problem.

As a closing remark on deep equality over graphs, notice that the precomputation does *not* subsume to subgraph isomorphism. This is due to the fact, that deep equal considers only two nodes together with all their respective descendants. General subgraph isomorphism considers any node induced subgraph of the target graph (or any connected subgraph for connected subgraph isomorphism).




(a) $Q_1^a$ (absolute value join) (b) $Q_1^b$ (relative value join)

(c) $Q_2^a$ (identity anti-join) (d) $Q_2^b$ (sibling order)

Figure 31. Exemplary Query Graphs

### 6.5.3 EXAMPLES

To illustrate, that the restriction to non-recursive CIQLOG (with all data graph relations) still yields many interesting queries, we discuss a few examples of non-recursive CIQLOG queries and, in a preview of Part III, show counterparts in Xcerpt or XQuery.

Let us first consider only the bodies of CIQLOG rules. Figure 31 shows graphical representations for four such CIQLOG rule bodies (or queries) against the data from Figure 24. This common, intuitive representation of queries as graphs is used throughout this paper: Query variables are represented as nodes with labels and values, as well as root nodes represented as in data graphs. Edges annotated with relation names represent atoms connecting query variables. Answer variables are marked by darker rectangles whereas normal variables are indicated by lighter circles.

The first query (Figure (a)) selects paper authors that are also members (of the pc) at a named conference. Although the visual representation is already graph-, rather than tree-shaped, this query can still be expressed in XPath (using abbreviated syntax for closure axis), which, if we disregard functions and equality, only expresses tree-shaped queries.

```
/conference//paper/author[text() =
    /conference[@title]/member/text()]
```

The following gives the textual representation of the query as a CIQLOG rule body (assuming only $v_3$ occurs in the head and is thus the only answer variable):

$$\begin{aligned}\longleftarrow\ & \mathfrak{L}(v_1, \text{conference}) \wedge \textsc{child}_+(v_1, v_2) \wedge \\ & \mathfrak{L}(v_2, \text{paper}) \wedge \textsc{child}(v_2, v_3) \wedge \mathfrak{L}(v_3, \text{author}) \wedge \textsc{value}(v_3, v_4) \wedge \\ & \mathfrak{L}(v_5, \text{conference}) \wedge \textsc{attr}(v_5, v_8) \wedge \mathfrak{L}(v_8, \text{title}) \wedge \textsc{child}_+(v_5, v_6) \wedge \\ & \mathfrak{L}(v_6, \text{member}) \wedge \textsc{value}(v_6,\ v_7) \wedge \cong(v_4,\ v_7).\end{aligned}$$

Path relations are abbreviated here since the label sets are singleton in all cases.

On the other hand, the second query (Figure (a)), though still unary, already requires a language with multiple variables such as XQuery or Xcerpt. Intuitively, this is the case so as to be able to express that matched bindings for $v_4$ and $v_6$ are connected to the same binding for $v_1$. In XQuery, e.g., $Q_1^b$ can be expressed as

```
for $c in /conference[@title], $a in $c//paper/author
where $c/member = $a
return $a
```

It selects an author only if he is also a (pc) member at the *same* conference. Where the difference between the two queries is quite large in XQuery, CIQLOG uses nearly the same queries omitting the label test for the second conference node and connecting all its children to $v_1$:

$$\begin{aligned}\longleftarrow\ & \mathfrak{L}(v_1, \text{conference}) \wedge \textsc{attr}(v_1, v_7) \wedge \mathfrak{L}(v_7, \text{title}) \wedge \textsc{child}_+(v_1, v_2) \wedge \\ & \mathfrak{L}(v_2, \text{paper}) \wedge \textsc{child}(v_2, v_3) \wedge \mathfrak{L}(v_3, \text{author}) \wedge \textsc{value}(v_3, v_4) \wedge \\ & \textsc{child}_+(v_1, v_5) \wedge \mathfrak{L}(v_5, \text{member}) \wedge \textsc{value}(v_5,\ v_6) \wedge \cong(v_4,\ v_6).\end{aligned}$$

Finally, the two queries $Q_2^a$ and $Q_2^b$ show how the same query intent (viz., *to select papers that are cited in two different papers at the same conference together with the name of that conference*) can be expressed differently: $Q_2^a$ uses a negated equality condition between to ensure that its two different papers, $Q_2^b$ uses order to ensure the same. However, the two queries are only equivalent if (a) CHILD edges outgoing from conferences are ordered and (b) if there are no multi-edges between conferences and papers (which is always the case if the data is tree shaped). Otherwise, $\ll_+$ is no longer necessarily anti-reflexive.

Both queries can be expressed only in languages with multiple variables. The following Xcerpt query term is equivalent to query $Q_2^a$ (up to representation of ID/IDREF links):

```
conference{{
 paper{{ cite{{
   var Paper → idvar cited @ paper{{ }} }} }}
 paper{{ cite{{
              idvar cited @ paper{{ }} }} }}
 name{{ var Name }} }}
```

Notice, that the anti-join is not expressed explicitly, but rather guaranteed by Xcerpt's injective mapping for sibling nodes (cf. [188] and Chapters 3 and 7). In CIQLog, we obtain:

$$
\begin{aligned}
\longleftarrow\ & \mathfrak{L}(v_1, \text{conference}) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_1, v_5) \wedge \\
& \quad \mathfrak{L}(v_2, \text{paper}) \wedge \text{ATTR}(v_2, v_3) \wedge \mathfrak{L}(v_3, \text{cite}) \wedge \text{REF}(v_3, v_4) \wedge \mathfrak{L} \\
& \qquad (v_4, \text{paper}) \wedge \\
& \quad \mathfrak{L}(v_5, \text{paper}) \wedge \text{ATTR}(v_5,\ v_6) \wedge \mathfrak{L}(v_6, \text{cite}) \wedge \text{REF}(v_6, v_7) \wedge \mathfrak{L} \\
& \qquad (v_7, \text{paper}) \wedge \\
& \quad \doteq(v_4,\ v_7) \wedge \neg(\ \cong(v_2, v_5)\ ) \wedge \text{ATTR}(v_1, v_8) \wedge \mathfrak{L}(v_8, \text{title}).
\end{aligned}
$$

Finally, $Q_2^b$ can be expressed in XQuery as follows:

```
for $c in /conference, $n in $c/name, $p1 in $c/paper,
   $cited = $p1/@cite->paper
where (some $p2 in $c/paper satisfies $p1 << $p2 and
      (some $cited2 in $p2/@cite->paper
            satisfies $cited is $cited2))
return ($cited, $name)
```

For CIQLog, the textual form is mostly as $Q_2^a$, but uses in line 4 $\ll_+$ instead of a negated identity equivalence. In CIQLog, the child relation between $v_1$ and $v_5$ is implied by the order relation between $v_2$ and $v_5$ and may be omitted (this can be achieved in XQuery by using the optional following axis).

$$
\begin{aligned}
\longleftarrow\ & \mathfrak{L}(v_1, \text{conference}) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_1, v_5) \wedge \\
& \quad \mathfrak{L}(v_2, \text{paper}) \wedge \text{ATTR}(v_2, v_3) \wedge \mathfrak{L}(v_3, \text{cite}) \wedge \text{REF}(v_3, v_4) \wedge \mathfrak{L} \\
& \qquad (v_4, \text{paper}) \wedge \\
& \quad \mathfrak{L}(v_5, \text{paper}) \wedge \text{ATTR}(v_5,\ v_6) \wedge \mathfrak{L}(v_6, \text{cite}) \wedge \text{REF}(v_6, v_7) \wedge \mathfrak{L} \\
& \qquad (v_7, \text{paper}) \wedge \\
& \quad \doteq(v_4,\ v_7) \wedge \ll_+(v_2,\ v_5) \wedge \text{ATTR}(v_1, v_8) \wedge \mathfrak{L}(v_8, \text{title}).
\end{aligned}
$$

Part III

# PRACTICE. CASE STUDIES: XCERPT, XQUERY, SPARQL

# 7 TRANSLATING XCERPT 2.0



## 7.1 INTRODUCTION

In Part I, we have introduced Xcerpt 2.0 as an example of a versatile Web query language and argued that versatility is increasingly becoming an essential requirement for Web queries. With CIQLOG we have the formal foundation to demonstrate how to translate queries in different languages and on different data formats into the same formal framework that can be evaluated using the CIQCAG algebra introduced in Part IV.

The following discussion of the translation of Xcerpt 2.0 starts with a fragment only, viz. non-recursive, single-rule Core Xcerpt which can be translated into a single non-recursive CIQLOG rule (and thus be evaluated by a single CIQCAG expression). We first define that fragment and its syntax in Section 7.2, then illustrate its semantics along examples in Section 7.3. The semantics is fully aligned with full Xcerpt. The actual translation of non-recursive, single-rule Xcerpt is covered in Section 7.4 and concluded by an outlook towards the translation of full Xcerpt in Section 7.5.

## 7.2 NON-RECURSIVE, SINGLE-RULE CORE XCERPT

We choose a fragment of the rule-based, Web and Semantic Web query language Xcerpt [188]. Before we characterize the language fragment, this chapter gives a brief recall of some of Xcerpt's most relevant features in the

context of this translation. For a proper introduction please see Chapter 3 and [188].

*Introducing Xcerpt*

An Xcerpt program consists of a finite set of Xcerpt *rules*. The rules of a program are used to derive new, or transform existing, data from existing data (i.e. the data being queried). *Construct rules* are used to produce intermediate results while *goal rules* form the output of programs.

While Xcerpt works directly on XML or RDF data, it has its own data format for modeling XML documents or RDF graphs, viz. Xcerpt *data terms*. For example, the XML snippet `<book><title>White Mughals</title></book>` corresponds to the data term `book [ title [ "White Mughals"] ]`. The data term syntax makes it easy to reference XML document structures in queries and extends XML slightly, most notably by allowing unordered data and making references first class citizens (thus moving from a tree to a proper graph data model).

For instance, in the following query the construct rule defines data about books and their authors which is then queried by the goal. Intuitively, the rules can be read as deductive rules (like in, say, Datalog): if the body (after **FROM**) holds, then the head (following **CONSTRUCT** or **GOAL**) holds. A rule with an empty body is interpreted as a fact, i.e., the head always holds.

```
GOAL
  authors [ var X ]
FROM
  book [[ author [ var X ] ]]
END

CONSTRUCT book [ title [ "White Mughals" ],
        author [ "William Dalrymple" ] ] END
```

*Terms*

Xcerpt *query terms* are used for querying data terms and intuitively describe patterns of data terms. Query terms are used with a pattern matching technique[1] to match data terms. Query terms can be configured to take partiality and/or ordering of the underlying data terms into account during matching (indicated by different types of brackets). Query terms may also contain (logic) variables. If so, successful matching with data terms results in variable bindings used by Xcerpt rules for deriving new data terms. Matching, for instance, against the XML snippet above the query term `book [ title [ var X ] ]` with results in the variable binding `{X/"White Mughals"}`. In addition to the query term types discussed in [187], we also consider non-injective ordered and unordered query terms indicated by three braces or brackets, respectively.

1 Called *simulation unification*. For details of this technique, please refer to [187].

*Construct terms* are essentially data terms with variables. The variable binding produced via query terms in the body of a rule can be applied to the construct term in the head of the rule in order to derive new data terms. For the example above we obtain the data term `authors [ "William Dalrymple"]` as result.

**Definition 7.1** (Non-recursive Single-rule Xcerpt)**.** Let $P$ be an Xcerpt program. Then, $P$ is a non-recursive, single-rule Xcerpt program, if it consists of a single **`GOAL`** rule and arbitrary many data terms.

In other words, there is a single rule whose body is evaluated a given set of data terms. If the body matches, the head is then evaluated given the bindings generated by the body.

### 7.2.1 FORMAL SYNTAX

In the following, we omit some of Xcerpt's more advanced features to allow for a translation that is reasonably compact and easy to follow. The most prominent omitted features are query terms involving **`optional`** or **`except`** and construct terms with **`some`**, **`first`**, declare blocks, order specifications, and conditions. Most of these limitations are purely for presentation reasons and the omitted constructs can easily be integrated in the translation presented here. This is, however, not the case for **`except`** which is not supported by non-recursive, single-rule CIQLog as it requires composition, i.e., first the sub-terms excluded by **`except`** are removed from a surrounding sub-graph binding, then that binding is used in the remainder of the query (e.g., for deep equality). This can be realized by multiple CIQLog rules, but not in non-recursive, single-rule CIQLog as defined in Chapter 6.4.

*Core Xcerpt*

We call the resulting language fragment non-recursive, single-rule *Core* Xcerpt, abbreviated as $\text{Xcerpt}^{\text{core,NR,SR}}$ and specify its full grammar in Figure 32 (using common EBNF-like notation).

The usual semantic restriction for Xcerpt rules apply, e.g., range restriction (each variable occurring in the head must also occur positively in each body disjunct where both **`not`** and **`without`** are considered as negative expressions); **`without`** may not occur in total query terms (indicated by single brackets or braces as in `title [var X]`); an identity variable can only occur with `idvar`, a label variable only in label and term position, but not with →, `idvar`, or `position`, a position variable only with `position`; **`without`** and **`position`** may not occur for top-level terms; each referenced term identifier is defined somewhere; no term identifier is defined twice.

*Limitations of the translation*

In the following, we assume a few additional restrictions to allow for a concise and easy to follow description of the translation: **(1)** We do not

⟨*rule*⟩ ::= 'GOAL' ⟨*cterm*⟩ 'FROM' ⟨*query*⟩ 'END'

⟨*cterm*⟩ ::= (⟨*identifier*⟩ '@')? ⟨*label*⟩ ⟨*hchildren*⟩ | '"' ⟨*string*⟩ '"'
| ⟨*reference*⟩ | ⟨*id-variable*⟩ | ⟨*variable*⟩
| ⟨*grouping*⟩

⟨*grouping*⟩ ::= 'all' ⟨*cterm*⟩ 'group-by' '(' ⟨*variable*⟩+ ')'

⟨*hchildren*⟩ ::= '{' ⟨*cterm*⟩* '}' | '[' ⟨*cterm*⟩* ']'

⟨*query*⟩ ::= 'and' '(' ⟨*query*⟩',' ⟨*query*⟩ ')'
| 'or' '(' ⟨*query*⟩',' ⟨*query*⟩ ')'
| 'not' '(' ⟨*query*⟩ ')'
| ⟨*qterm*⟩

⟨*qterm*⟩ ::= ⟨*term-id*⟩? ⟨*position*⟩? ⟨*label*⟩ ⟨*qchildren*⟩ | '"' ⟨*string*⟩ '"'
| ⟨*reference*⟩
| ⟨*variable*⟩ ('→'⟨*qterm*⟩)?
| ('desc' | 'without') ⟨*qterm*⟩

⟨*term-id*⟩ ::= (⟨*identifier*⟩ | ⟨*id-variable*⟩) '@'

⟨*reference*⟩ ::= '^' ⟨*identifier*⟩

⟨*position*⟩ ::= 'position' (⟨*number*⟩ | ⟨*variable*⟩)

⟨*qchildren*⟩ ::= '{' (⟨*qterm*⟩ ',')* '}' | '[' (⟨*qterm*⟩ ',')* ']'
| '{{' (⟨*qterm*⟩ ',')* '}}' | '[[' (⟨*qterm*⟩ ',')* ']]'
| '{{{' (⟨*qterm*⟩ ',')* '}}}' | '[[[' (⟨*qterm*⟩ ',')* ']]]'

⟨*variable*⟩ ::= 'var' ⟨*identifier*⟩

⟨*id-variable*⟩ ::= 'idvar' ⟨*identifier*⟩

⟨*label*⟩ ::= ⟨*variable*⟩ | ⟨*identifier*⟩ | '*'

Figure 32. Syntax of non-recursive, single-rule Core Xcerpt

```
conference [
 title [ "Storage Media" ] date [ "44 B.C." ]
 p1 @ paper [ title [ "Wax Tablets" ]
  ^p2 author [ "Cicero" ]
 ]
 p2 @ paper [
  ^p1 author [ "Hirtius" ]
 ]
 pc[ member[ "Cicero" ] member[ "Atticus" ] ]
]
```

Figure 33. Xcerpt$^{core,NR,SR}$ dataterm on conferences and papers

consider nested grouping lists as in

```
all a[var X, all (var Y, d)]
```

**(2)** We consider node injectivity instead of position injectivity for matching injective term specifications such as `a{{b, var X}}` or `a{b, var X}` where `b` and `var X` must be connected by edges with different edge positions to a in standard Xcerpt, where we consider that `var X` binds only to different nodes than `b`. **(3)** We consider `*` as abbreviation for the Xcerpt regular expression `/.*/` that indicates that the label of a query term may be arbitrary. **(4)** We omit XML specificities present in full Xcerpt terms for such as comments, processing-instructions, attributes, namespaces. **(5)** We normalize grouping expressions (`all ... group-by`) such that all free variables in each grouping expression are listed in the `group-by` clause of that grouping. In full Xcerpt, this is not necessary but possible and yields a grouping expression with the same semantics.

*Xcerpt$^{core,NR,SR}$ examples*

Before we turn to a more detailed examination of the semantics of Xcerpt query and construct terms in the following section, we give a first intuition by the following two examples on a slight simplification of the data introduced in Section 5.5. Figure 33 shows a Core Xcerpt data term where namespaces and comments are removed and attributes are changed to elements, if compared to the data term from Section 5.5.

On that data term, the Xcerpt$^{core,NR,SR}$ rule in Figure 34 selects papers containing "Cicero" as author and "puts them in a shelf".

To illustrate the difference between references and copies of query terms consider the final two rules and their query terms shown in Figure 35: the left query term returns all papers such that there is another paper with at least one (structurally) same author, the right hand returns only papers where the identical author term is used in both cases (rather than just a

```
GOAL
  shelf{ all var X group-by(var X) }
FROM
  conference{{
    var X → paper{{
      desc author{{ "Cicero" }} }} }}
END
```

Figure 34. Xcerpt$^{\text{core,NR,SR}}$ rule to extract Cicero's papers to a shelf

```
GOAL
  shelf{ all var X group-by(var
      X) }
FROM
  conference{{
    var X → paper{{
      author{{
        var Y
    }} }}
    paper{{ author{{ var Y }} }}
        }}
END
```

```
GOAL
  shelf{ all var X group-by(var
      X) }
FROM
  conference{{
    var X → paper{{
      idvar A @ author{{
        var Y
    }} }}
    paper{{ idvar A @ author }} }}
END
```

Figure 35. Structural versus identity equivalence in Xcerpt

structurally equivalent one).

## 7.3 XCERPT SEMANTICS BY EXAMPLE

An Xcerpt term is essentially a labeled list of children. In addition to the label, we also record whether a term is *ordered* and, if it is a query term, if it is *total* or *partial injective* or *partial non-injective*. Recall, that a query term specifies a query by exemplifying (think QBE [205]) the shape of the matched data terms. However, to be effective, we leave out certain parts of that shape and focus only on the parts relevant to the query intent. Leaving out certain parts is achieved by various forms of incompleteness: regarding the structural relations by moving from child to descendant relations (indicated by the desc keyword), regarding how complete the list of given children is using the three latter properties above: If a query term is total, there may be no children of a matching query term in addition

to the ones matched by each of the query term's children. If partial, there may be additional ones. If the query term is in addition injective, each of its children is mapped to a unique child of a matching data term. We denote ordered query terms with square brackets, otherwise we use curly braces. For total terms we use single such brackets, for partial injective double, for partial non-injective triple.

The data term on conferences in Figure 33 serves as an example for total terms since data (and construct) terms only contain this term type. Query terms also contain partial terms as evidenced by the query examples in Figures 34 and 35.

| | Query term | Data terms |
|---|---|---|
| T1 | a{ } | ≼ a{ }; a[ ]<br>⋠ b{ } |
| T2 | a[ ] | ≼ a[ ]<br>⋠ b{ }; a{ } |
| T3 | a{ b } | ≼ a{ b }<br>⋠ a{ b, b } |
| T4 | a{ b, b } | ≼ a{ b, b }<br>⋠ a{ b }; a{ b, b, b } |
| P1 | a{{ b }} | ≼ a{ b }; a{ c, b, d }; a{ b, b }<br>⋠ a{ }; |
| P2 | a[[ b, c ]] | ≼ a[ b, c ]; a[ d, b, e, c ]<br>⋠ a[ c, b ]; a{ b, c } |
| I1 | a{{{ b, b }}} | ≼ a[ b ]; a{ c, b, d }; a{ b, b }<br>⋠ a[ ]; a{ } |
| I2 | a[[[ b, b, d ]]] | ≼ a[ b, d ]; a[ c, b, d ];<br>⋠ a[ d, b ]; a{ } |
| D1 | a{ desc b } | ≼ a[ b ]; a[ c{ b, e } ];<br>⋠ a{ d, c{ b } }; |
| D2 | a{ desc b, desc c } | ≼ a[ b, e[ c ] ];<br>⋠ a{ b, c, d }; a{ e[b, c] }; |
| D3 | a{{ desc b, desc c }} | ≼ a[ b, e[ c ] ]; a{ b, c, d };<br>⋠ a{ e[b, c] }; |
| D4 | a{{{ desc b, desc c }}} | ≼ a[b,e[c]]; a{b,c,d}; a{e[b,c]}; |
| W1 | a{{ b, without( c ), d }} | ≼ a[ b, d ];<br>⋠ a{ b, c, d }; a{ c, b, d }; |
| W2 | a[[ b, without( c ), d ]] | ≼ a[ b, d ]; a[ c, b, d ];<br>⋠ a[ b, c, d ]; |
| W3 | a[[ b, without( c, d ),<br>e ]] | ≼ a[ b, e ]; a[ b, c, e ];<br>⋠ a[ b, c, d, e ]; |
| W4 | a[[ b, without( c ),<br>without( d ), e ]] | ≼ a[ b, e ];<br>⋠ a[ b, c, e ]; a[ b, c, d, e ]; |

Table 9. Query terms and matching data (; separates different data terms)

MATCHING QUERY TERMS. Table 9 illustrates how these properties affect the matching of query terms against data terms by example. For space reasons, we omit in query terms empty double braces and in data

term empty single braces, i.e., c reads c{{ }} in a query term and c{ } in a data term. We denote matching using $\preccurlyeq$ (simulation unification from in [187], but here we consider only data terms on the right hand), non matching with $\not\preccurlyeq$.

The first examples T1–T4 illustrate matching of ordered and unordered total query terms. Note, that unordered query terms match against ordered data terms (since the use of the curly braces indicates only that we do not care about the order). In total query terms both terms have exactly the same number of children in all cases. This is what sets partial query terms (P1–P2, I1–I2) apart from total query terms. Here, we may have additional query terms in the data that are ignored. For partial non-injective query terms (I1–I2), two children of the query may even match to the same data term.

The remaining examples of Table 9 illustrate the two query term modifiers, `desc` and `without`. The former allows matching at any depth (cf. D1–D4). Totality and injectivity are still enforced between the children of a matching data term (observe the difference between D2, D3, and D4). The latter forbids the existence of a data term matching its enclosed query terms, cf. W1–W4. It may even take a list of query terms, in which case (W3) the query fails, if the entire list (and not just some subset of it) fails. This contrasts with the use of multiple `without`s as in W4.

| | Query term | Data terms | Bindings |
|---|---|---|---|
| V1 | `a{ var X }` | $\preccurlyeq$ `a[ b ]`;<br>$\not\preccurlyeq$ `a{ }`; `a[ b, c ]` | $\{X/\mathtt{b}\}$ |
| V2 | `a{{ var X }}` | $\preccurlyeq$ `a[ b, c ]`;<br>$\not\preccurlyeq$ `a{ }`; | $\{X/\mathtt{b}, X/\mathtt{c}\}$ |
| V3 | `a{{ var X, var X }}` | $\preccurlyeq$ `a[ b, b ]`; `a{ c, b, b, d }`<br>$\not\preccurlyeq$ `a{ b, c }`; `a{ b }` | $\{X/\mathtt{b}_1, X/\mathtt{b}_2\}$ |
| V4 | `a{{{ var X, var X }}}` | $\preccurlyeq$ `a[ b, b ]`; `a{ c, b, b, d }`<br>$\preccurlyeq$ `a{ b }`<br>$\not\preccurlyeq$ `a{ b, c }`; | $\{X/\mathtt{b}_1, X/\mathtt{b}_2\}$<br>$\{X/\mathtt{b}\}$ |
| V5 | `a{ var X{ var X } }` | $\preccurlyeq$ `a[ b{ "b"} ]`;<br>$\not\preccurlyeq$ `a{ b, c }`; | $\{X/\text{“b”}\}$ |
| V6 | `a{ var X → c, var X }` | $\preccurlyeq$ `a[ c, c ]`;<br>$\not\preccurlyeq$ `a{ b, b }`; | $\{X/\mathtt{b}\}$ |
| V7 | `a{ desc var X }` | $\preccurlyeq$ `a[ c{ b, e[ f ] } ]`;<br>$\not\preccurlyeq$ `a{ d, c{ b } }`; | $\{X/\mathtt{c}\{\ldots\}, X/\mathtt{b}, X/\mathtt{e}[\ldots], X/\mathtt{f}\}$ |
| V8 | `a{{var X, without(var X)}}` | $\preccurlyeq$ `a[ b ]`; `a[ b, c ]`;<br>$\not\preccurlyeq$ `a{ b, b }`; `a[ b, b ]`; | $\{X/\mathtt{b}\}$ |

Table 11. Query terms containing variables and their bindings

The last remaining feature of query terms are variables, the effect of which on term matching is illustrated in Table 11: Essentially, a variable matches any single term (or label, or position, or node, if so placed), but matches are recorded in the bindings of the query. If a variable occurs multiple times (V3), the matched query terms must be structurally equivalent (deep equal, cf. Section 5.6.6). A variable may occur as a label (V5), in which case it is bound to the value of the label and can only match with other labels or character data (as the “b” in V5). A variable may occur as in a term restriction before → (V6), in which case the right hand query term restricts the matching bindings for $X$. Finally, it can be combined with `desc` and `without` with the expected result (V7, V8).

INSTANTIATING CONSTRUCT TERMS. Once the body of a rule is matched against the input data, the bindings for the answer variables can be used to instantiate the construct term of an Xcerpt$^{\text{core,NR,SR}}$ rule. Again, we illustrate the semantics of construct terms along a number of examples, cf. Table 13, using the following binding tuples for answer variables X and

| | Construct term | Result data |
|---|---|---|
| C1 | `a{ b, c }` | $\xrightarrow{\varepsilon}$ `a{ b, c }` |
| C2 | `a{ id @ b, ^id }` | $\xrightarrow{\varepsilon}$ `a{ id' @ b, ^id' }` |
| C3 | `a{ var X }` | $\xrightarrow{\varepsilon}$ $\mathtt{a\{\ b_1\ \}}$ |
| C4 | `a{ var X, var Y }` | $\xrightarrow{\varepsilon}$ $\mathtt{a\{\ b_1, c_1\ \}}$ |
| G1 | `a{ all var X group-by(var X)}` | $\xrightarrow{\varepsilon}$ $\mathtt{a\{\ b_1, b_2, b_3[e,\ f]\ \}}$ |
| G2 | `all a{ var X group-by(var X)}` | $\xrightarrow{\varepsilon}$ $\mathtt{a\{\ b_1\ \}, a\{\ b_2\ \}, a\{\ b_3[e,\ f]\ \}}$ |
| G3 | `all a{ all var X group-by(var X), var Y } group-by(var Y)` | $\xrightarrow{\varepsilon}$ $\mathtt{a\{\ b_1, b_2, b_3[e,\ f], c_1\ \}}$, $\mathtt{a\{\ b_1, b_3[e,\ f], c_2\ \}}$ |
| G4 | `all a{ } group-by(var Y)` | $\xrightarrow{\varepsilon}$ `a{ }, a{ }` |

Table 13. Construct terms and their instantiation

Y:

$$\mathcal{B} = \{\{\mathtt{X}/b_1, \mathtt{Y}/c_1\}, \{\mathtt{X}/b_2, \mathtt{Y}/c_1\}, \{\mathtt{X}/b_3[\mathtt{e},\ \mathtt{f}], \mathtt{Y}/c_1\},\\ \{\mathtt{X}/b_1, \mathtt{Y}/c_2\}, \{\mathtt{X}/b_3[\mathtt{e},\ \mathtt{f}], \mathtt{Y}/c_2\}\}$$

If a construct term contains no variables (C1–C2), the only resulting data term has the exact same shape, possibly renaming local identifiers for references (C2). If it contains variables outside grouping expressions (C3, C4) these are instantiated by some of their bindings (we choose here the first binding). Grouping expressions iterate over the bindings of their grouping variables and instantiate their contained construct term once for each binding tuple of the grouping variables. The scope of the grouping expression defines, in this case which parts of the construct term are repeated (G1, G2). It is not necessary that the contained construct terms actually contain the occurrences of the grouping variables (G4), though that is usually the case. Employing nested grouping terms as in G3, we can create complex nestings of related bindings, here, e.g., for each binding of Y the corresponding bindings of X are grouped as siblings.

## 7.4 TRANSLATING NON-RECURSIVE CORE XCERPT

With the syntax and intuitive semantics of Xcerpt$^{\text{core,NR,SR}}$ established, we can turn to the actual translation. The translation is split in three parts, the translation of construct terms to CIQLOG rule heads, the translation of query

terms to CIQLOG rule bodies, and the "glue", the translation of Xcerpt$^{\text{core,NR,SR}}$ rules to CIQLOG rules. We start off with the translation of rules and a "grand example" that illustrates the principles of the translations. The details of construct and query term translation are discussed in the remainder of this section.

### 7.4.1 RULES

Recall, that a Xcerpt$^{\text{core,NR,SR}}$ program consists of a single Core Xcerpt **GOAL** rule. Such a rule is translated by the $\text{tr}_{\text{Xcerpt}}$ translation function: as follows:

$$\text{tr}_{\text{Xcerpt}}\langle\textbf{GOAL}\ \mathit{head}\ \textbf{FROM}\ \mathit{query}\ \textbf{END}\rangle = C \longleftarrow Q \quad \textbf{where}\ (\mathcal{E}, Q) = \text{tq}\langle\mathit{body}\rangle,\ C = \text{tc}(\mathcal{E})\langle\mathit{head}\rangle$$

It translates the body first, redirecting the resulting environment containing associations of Xcerpt (answer) variables to CIQLOG variables to the translation of the head of the input rule. Finally, the translation of the body and head are combined into the translation of the full rule. The translation functions for head and body, tc and tq, are defined below in Sections 7.4.2 and Section 7.4.3. Here and in the following, we denote the set of common edge labels with $L = \{\text{CHILD}, \text{COMMENT}, \text{VALUE}\}$.

EXAMPLES. Before turning to the precise definitions of those translation functions, let us return to the example data and rule from Figures 33 and 34.

Xcerpt data terms can be translated to CIQLOG (thus giving a formal definition for the mapping described in Section 5.5) by the translation function for Xcerpt rule heads using an empty environment as input. The environment can be empty since it is responsible only for passing Xcerpt to CIQLOG variable mappings and data terms contain no Xcerpt variables.

The data term $D$ in Figure 33 is translated by $\text{tc}(\varnothing)\langle D\rangle$ into the following CIQLOG rule:

```
root(id1) ∧ 𝔏(id1, conference) ∧ 𝔇^L(id1) ∧ CHILD(id1, id2, order(⊤,1)) ∧ CHILD
    (id1, id3, order(⊤,2)) ∧ CHILD(id1, id4, order(⊤,3)) ∧ CHILD(id1, id5,
    order(⊤,4)) ∧ CHILD(id1, id6, order(⊤,5)) ∧
  𝔏(id2, title) ∧ 𝔇^L(id2) ∧ VALUE(id2, id21, order(⊤,1)) ∧ 𝔏(id21, "Storage
    Media") ∧
  𝔏(id3, date) ∧ 𝔇^L(id3) ∧ VALUE(id3, id31, order(⊤,1)) ∧ 𝔏(id31, "44 B.C.") ∧
  𝔏(id4, paper) ∧ 𝔇^L(id4) ∧ CHILD(id4, id41, order(⊤,1)) ∧ CHILD(id4, id5,
    order(⊤,2)) ∧ CHILD(id4, id42, order(⊤,3)) ∧
    𝔏(id41, title) ∧ 𝔇^L(id41) ∧ VALUE(id41, id411, order(⊤,1)) ∧ 𝔏(id411, "Wax
      Tablets") ∧
    𝔏(id42, author) ∧ 𝔇^L(id42) ∧ VALUE(id42, id421, order(⊤,1)) ∧ 𝔏(id421,
      "Cicero") ∧
```

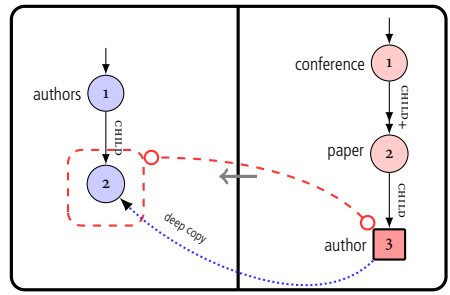



Figure 36. Resulting CIQLog rule

```
    𝔘(id5, paper) ∧ 𝔒^L(id5) ∧ CHILD(id5, id4, order(⊤,1)) ∧ CHILD(id5, id51,
        order(⊤,2)) ∧
8       𝔘(id51, author) ∧ 𝔒^L(id51) ∧ VALUE(id51, id511, order(⊤,1)) ∧ 𝔘(id511,
            "Hirtius") ∧
    𝔘(id6, pc) ∧ 𝔒^L(id6) ∧ CHILD(id6, id61, order(⊤,1)) ∧ CHILD(id6, id62,
        order(⊤,2)) ∧
10      𝔘(id61, member) ∧ 𝔒^L(id61) ∧ VALUE(id61, id611, order(⊤,1)) ∧ 𝔘(id611,
            "Cicero") ∧
        𝔘(id62, member) ∧ 𝔒^L(id62) ∧ VALUE(id62, id621, order(⊤,1)) ∧ 𝔘(id621,
            "Atticus") ⟵ true.
```

The relations are all ordered as the data term is deliberately similar to the XML fragment from Section 5.3 and thus contains only ordered terms. However, in general, Xcerpt data terms may also contain unordered terms.

Here, and in the following we use two abbreviations for head formulas:

```
1   child(id1(x⃗1),id2(x⃗2),o)
```

where $o$ is some order term and $\vec{x}_1, \vec{x}_2$ are variable lists, abstracts the edge construction necessary in CIQLog and thus is an abbreviation for (with $\mathsf{id}_N$ a new identifier):

```
1   source(id1(x⃗1),idN(x⃗2)),, sink(id2(x⃗2),idN(x⃗2)),,
        position(idN(x⃗2),o))
```

In the same way, $\mathtt{child}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_2(\vec{x}_2))$ is an abbreviation for ((with $\mathsf{id}_N, \mathsf{id}_M$ new identifiers):

```
source(id1(x⃗1),idN(x⃗2)),, sink(id2(x⃗2),idN(x⃗2)),,
    position(idN(x⃗2),idM(x⃗2))
```

To illustrate the full translation, first consider the following very simple Xcerpt rule selecting all authors of papers and grouping them under a new root authors:

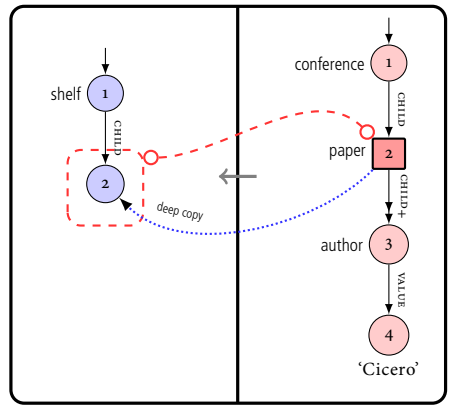



Figure 37. CIQLog rule for Xcerpt program from Figure 34

```
GOAL
  authors{ all var X group-by(var X) }
FROM
  conference{{ desc paper{{ var X → author }} }}
END
```

If we translate this Xcerpt rule to CIQLog we obtain the query visualized in Figure 36 (using the visualization from Chapter 6 and additionally depicting the scope of a grouping variable by red rectangles ⬚ as well as query variables used in the head by directed connections ⋯▸).

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1, \texttt{authors}) \wedge \textsc{child}(\mathsf{id}_1, \mathsf{id}_2(v_3)) \wedge \textsc{deep-copy}(v_3, \mathsf{id}_2(v_3)) \longleftarrow \texttt{root}(v_1) \wedge \mathfrak{L}(v_1, \texttt{conference}) \wedge \textsc{child}_+(v_1, v_2) \wedge \mathfrak{L}(v_2, \texttt{paper}) \wedge \textsc{child}(v_2, v_3) \wedge \mathfrak{L}(v_3, \texttt{author}).$$

Finally, we reconsider the example rule from Figure 34, an Xcerpt rule querying that data and selecting all papers with author "Cicero" and "puts them on a shelf". Applying $\mathrm{tr}_{\text{Xcerpt}}$ to that rule yields the CIQLog program depicted in Figure 37 and given in textual form in the following:

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1, \texttt{shelf}) \wedge \textsc{child}(\mathsf{id}_1, \mathsf{id}_2(v_2)) \wedge \textsc{deep-copy}(v_2, \mathsf{id}_2(v_2)) \longleftarrow \texttt{root}(v_1) \wedge \mathfrak{L}(v_1, \texttt{conference}) \wedge \textsc{child}(v_1, v_2) \wedge \mathfrak{L}(v_2, \texttt{paper}) \wedge \textsc{child}_+(v_2, v_3) \wedge \mathfrak{L}(v_3, \texttt{author}) \wedge \textsc{value}(v_3, v_4), \mathfrak{L}(v_4, \texttt{"Cicero"}).$$

### 7.4.2 CONSTRUCT TERMS

With the intuition from the above examples, the translation of construct terms using tc can be discussed in more detail. The full specification of tc and its helper functions $\mathrm{tc}_{term}$, for translating actual construct terms, and

$\text{tc}_{label}$, for translating labels of construct terms, is given in Table 16. We normalize construct terms such that there are no variables outside (**all**) grouping terms. If there are such variables we wrap an **all** around the entire construct term with the all variables outside any other **all** as grouping variables. For instance, `a{var X}` becomes `all a{var X} group-by(var X)`.

tc adds a root atom and then delegates the translation to $\text{tc}_{term}$. This allows us to translate all query terms in the same way, yet yields the necessary root atom for top-level query terms. $\text{tc}_{term}$ is called with the environment $\mathcal{E}$ as only parameter which is passed to tc by $\text{tr}_{\text{Xcerpt}}$ and contains the mappings form Xcerpt variables to CIQLog query variables in the translation of the rule body. We write $\mathcal{E} +\!\!+ (\text{X}, v)$ to add a mapping from the Xcerpt variable X to the CIQLog variable $v$ into $\mathcal{E}$. In addition to those mappings, we let the environment contain the current grouping (or iteration) variables, a sequence accessed as $\mathcal{E}$.iter. We use list concatenation ($\circ$) to append variables to $\mathcal{E}$.iter. $\text{tc}_{term}$ returns a pair $(C, v)$ where $C$ is the CIQLog formula resulting from the translation of the passed Xcerpt expression and $v$ the variable associated with the top-level node in the translation of the expression.

For each kind of construct term, there is a matching rule for $\text{tc}_{term}$ in Table 16. The translation of most cases is fairly straightforward. The most involved cases are the structure terms (the first two cases) where we distinguish ordered and unordered terms. For both, we translate the child terms and connect their top-level variables to the variable of the current term by either CHILD or VALUE. The label is translated using the helper function $\text{tc}_{label}$ (which distinguishes between the translation of plain labels and of variables). Strings are translated like terms with empty, unordered term lists (case 3); references (like the term identifier part of a structured term) by retrieving an existing CIQLog variable for the Xcerpt identifier *tid* or creating a new one and storing that mapping in the environment $\mathcal{E}$ (case 4). Standard variables are translated using deep-copy, id-variables by simply retrieving the mapped query variable (recall that rules are range restricted and thus such a binding always exists), case 5–6. Finally, grouping terms are translated by adding the grouping variables to the sequence of iteration variables used for the translation of all contained construct terms, case 7.

EXAMPLES. We conclude the illustration of the translation function for construct terms by a collection of construct terms from Table 13 together with their CIQLog translation. Table 18 shows the translation to CIQLog for some of these construct terms. For convenience, we use $X$ and $Y$ to denote the query variables mapped to $X$ and $Y$ in a given environment $\mathcal{E}$, i.e., $\mathcal{E}(X)$ and $\mathcal{E}(Y)$. Recall, that the result is an expression containing some query variables that are replaced with their bindings for each binding tuple in turn.

| function | term | CIQLOG expression |
|---|---|---|
| $\mathrm{tc}(\mathcal{E})\langle \mathit{cterm} \rangle$ | | $= \texttt{root}(v) \wedge C$ **where** $(\mathcal{E}', C, v) = \mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \mathit{cterm} \rangle$ |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \mathit{tid} \texttt{ @ } \mathit{label}\{t_1, \ldots, t_k\} \rangle$ | | $= (\mathcal{E}_k, r_1(v, n_1) \wedge \ldots \wedge r_k(v, n_k) \wedge L \wedge C_1 \wedge \ldots \wedge C_k, v)$<br>**where** $v = \mathcal{E}(\mathit{tid})$ if defined, otherwise $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier<br>$L = \mathrm{tc}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{label} \rangle$<br>$r_i =$ VALUE if $t_i$ is "*string*", $r_i =$ CHILD otherwise<br>$\mathcal{E}_0 = \mathcal{E} +\!\!+ (\mathit{tid}, v) \qquad (\mathcal{E}_i, C_i, n_i) = \mathrm{tc}_{\mathit{term}}(\mathcal{E}_{i-1})\langle t_i \rangle$ |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \mathit{tid} \texttt{ @ } \mathit{label}[t_1, \ldots, t_k] \rangle$ | | $= (\mathcal{E}_k, r_1(v, n_1, \mathrm{order}(\top, 1, \mathcal{E}.\mathrm{iter})) \wedge \ldots \wedge r_k(v, n_k, \mathrm{order}(\top, k, \mathcal{E}.\mathrm{iter})) \wedge \mathfrak{O}(v, \text{CHILD}) \wedge L \wedge C_1 \wedge \ldots \wedge C_k, v)$<br>**where** $v = \mathcal{E}(\mathit{tid})$ if defined, otherwise $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier<br>$r_i =$ VALUE if $t_i$ is "*string*", $r_i =$ CHILD otherwise<br>$L = \mathrm{tc}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{label} \rangle$<br>$\mathcal{E}_0 = \mathcal{E} +\!\!+ (\mathit{tid}, v) \qquad (\mathcal{E}_i, C_i, n_i) = \mathrm{tc}_{\mathit{term}}(\mathcal{E}_{i-1})\langle t_i \rangle$ |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \text{"}\mathit{string}\text{"} \rangle$ | | $= (\mathcal{E}, L, v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier<br>$L = \mathrm{tc}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{string} \rangle$ |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \texttt{\^{}}\mathit{tid} \rangle$ | | $= (\mathcal{E} +\!\!+ (\mathit{tid}, v), \top, v), v)$ **where** $v = \mathcal{E}(\mathit{tid})$ if defined, otherwise<br>$v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ with id new identifier |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \texttt{var X} \rangle$ | | $= (\mathcal{E}, \text{deep-copy}(\mathcal{E}(X), v), v)$ **where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id new identifier |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \texttt{idvar X} \rangle$ | | $= (\mathcal{E}, \top, \mathcal{E}(\mathrm{X}))$ |
| $\mathrm{tc}_{\mathit{term}}(\mathcal{E})\langle \texttt{all } t \texttt{ group-by}(X_1, \ldots, X_n) \rangle$ | | $= \mathrm{tc}_{\mathit{term}}(\mathcal{E}')\langle t \rangle$ **where** $\mathcal{E}'$ = $\mathcal{E}$ with $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ [\mathcal{E}(X_1), \ldots, \mathcal{E}(X_n)]$ |
| $\mathrm{tc}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{label} \rangle$ | | $= \mathfrak{L}(v, \mathit{label})$ |
| $\mathrm{tc}_{\mathit{label}}(\mathcal{E}, v)\langle \texttt{var X} \rangle$ | | $= \cong (v, \mathcal{E}(\mathrm{X}))$ |

Table 16. Translating Xcerpt construct terms

| | Construct term | | cIQLog expression |
|---|---|---|---|
| C1 | `a{ b, c }` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1) \wedge \mathfrak{L}(\mathrm{id}_1,\texttt{a}) \wedge \textsc{child}(\mathrm{id}_1,\mathrm{id}_2) \wedge \mathfrak{L}(\mathrm{id}_2,\texttt{b}) \wedge \textsc{child}(\mathrm{id}_1,\mathrm{id}_3) \wedge \mathfrak{L}(\mathrm{id}_3,\texttt{c})$ |
| C2 | `a[ id @ b, ^id ]` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1) \wedge \mathfrak{L}(\mathrm{id}_1,\texttt{a}) \wedge \mathfrak{O}^L(\mathrm{id}_1) \textsc{child}(\mathrm{id}_1,\mathrm{id}_2,\texttt{order}(\top,1)) \wedge \mathfrak{L}(\mathrm{id}_2,\texttt{b}) \wedge \textsc{child}(\mathrm{id}_1,\mathrm{id}_2,\texttt{order}(\top,2))$ |
| C3 | `a{ var X }` | $\xrightarrow{tc}$ | normalized to G2. |
| G1 | `a{ all var X group-by(var X)}` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1) \wedge \mathfrak{L}(\mathrm{id}_1,\texttt{a}) \wedge \textsc{child}(\mathrm{id}_1,\mathrm{id}_2(X)) \wedge \text{deep-copy}(X,\ \mathrm{id}_2(X))$ |
| G2 | `all a{ var X group-by(var X)}` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1(X)) \wedge \mathfrak{L}(\mathrm{id}_1(X),\texttt{a}) \wedge \textsc{child}(\mathrm{id}_1(X),\mathrm{id}_2(X)) \wedge \text{deep-copy}(X,\ \mathrm{id}_2(X))$ |
| G3 | `all a{ all var X group-by(var X), var Y } group-by(var Y)` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1(Y)) \wedge \mathfrak{L}(\mathrm{id}_1(Y),\texttt{a}) \wedge \textsc{child}(\mathrm{id}_1(Y),\mathrm{id}_2(Y,\ X)) \wedge \text{deep-copy}(X,\ \mathrm{id}_2(Y,\ X)) \wedge \textsc{child}(\mathrm{id}_1(Y),\mathrm{id}_3(Y)) \wedge \text{deep-copy}(Y,\ \mathrm{id}_3(Y))$ |
| G4 | `all a{ } group-by(var Y)` | $\xrightarrow{tc}$ | $\texttt{root}(\mathrm{id}_1(Y)) \wedge \mathfrak{L}(\mathrm{id}_1(Y),\texttt{a})$ |

Table 18. Construct terms and their cIQLog translations

### 7.4.3 QUERIES AND QUERY TERMS

Bodies of Xcerpt rules are translated using tq. Again, we us an environment $\mathcal{E}$ to record already established mappings between Xcerpt and cIQLog variables. For queries we do not record iteration or grouping variables as there are no grouping expression in queries. The environment additionally contains isLabel(X) and isTerm(X, $v$) terms for Xcerpt variables X and cIQLog variables $v$. The former indicates that X is known to have occurred as a label variable. The latter that X has occurred in term position and is represented by $y$ in the cIQLog expression. These are used to establish proper variable occurrences in label and term position. As before, we use ⧺ to add these terms to a given environment $\mathcal{E}$. We do use an additional helper structure for the translation of query terms, viz. $\mathcal{V} = (v_\uparrow, V_\leftarrow, V_\rightarrow, r_\uparrow, r_\leftarrow, r_\rightarrow)$, and denote each component by $\mathcal{V}.V_\leftarrow$, etc. and with $()$ the empty $\mathcal{V}$. $\mathcal{V}$ holds the cIQLog variable for the parent term of the term to be translated ($v_\uparrow$), the variables for left and right siblings of that term ($V_\leftarrow$ and $V_\rightarrow$, respectively)

and the relations to these variables ($r_\uparrow$ for parent, $r_\leftarrow$ for left siblings, $r_\rightarrow$ for right siblings).

An Xcerpt rule body is first translated using tq which takes care of all top-level disjunction, conjunction, or negations. Note, that for disjunctions and conjunctions we propagate the environment return by the translation of the first operand ($\mathcal{E}_1$ in case 2 and 3) to the translation of the second operand. Thus Xcerpt variables occurring in both disjuncts are mapped to the same CIQLog variable. This assumes that, as usual, non-answer variables are standardized apart for each disjunct. After translating any top-level expressions, we turn to the evaluation of the top-level query terms which we root and translate using $\mathsf{tq}_{term}$ (case 5 of $\mathsf{tr}_{\mathsf{Xcerpt}}$). $\mathsf{tq}_{term}$ carries, in addition to the Xcerpt term to be translated, three parameters: the environment $\mathcal{E}$, a CIQLog variable for the current term (which may be $\perp$ indicating that no variable has been allocated for the term yet), and the context variables $\mathcal{V}$. Initially, there is no CIQLog variable for the current term yet (and thus the second parameter is $\perp$) and also no context variables $\mathcal{V}$ as a top-level Xcerpt term has no parent or sibling terms.

As for $\mathsf{tc}_{term}$, $\mathsf{tq}_{term}$ is specified by one case for each of the possible query terms. Cases 1–6 cover "prefixes" of full query terms, e.g., term identifiers, term restrictions, position specifications, or descendant modifiers. Cases 7–10 cover the four basic query term kinds, structured (case 7), character data (case 8), reference (case 9), and variable (case 10). Finally, case 11 covers `without` with may contain an entire term list and is thus a bit of the "odd man out" in the translation of query terms.

For $\mathsf{tq}_{term}$, we use five helper functions depicted in Table 22. **(1)** Bottom up, $\mathsf{tq}_{struct}$ creates a CIQLog expression for all the relations in a given $\mathcal{V}$ with a given current variable $v_{curr}$. It is used for the translation of each of the base term kinds (case 7–10 of $\mathsf{tq}_{term}$). **(2)** $\mathsf{tq}_{var}$ turns a Xcerpt variable into a CIQLog variable using the specified equivalence relation eq. If the Xcerpt variable is already defined in the current environment the stored CIQLog variable is retrieved and an equivalence atom is emitted, otherwise we simply record a new mapping from the given Xcerpt X variable to the CIQLog variable $v$. A special treatment of label variables as in the translation of construct terms is provided. **(3)** $\mathsf{tq}_{label}$ translates just the label of a structured term: if it is the wildcard $*$, no relation is added (any node fulfills that restriction), if it is a proper label a corresponding label relation is omitted, if it is variable a similar translation as for $\mathsf{tq}_{var}$ takes place but we mark the variable by isLabel in the environment to ensure that other occurrences of that variable are connected using $\cong$ and no other equivalence relation. **(4)** $\mathsf{tq}_{child}$ translates child lists of a structure term selected by the CIQLog variable $v_\uparrow$. Basically, it distinguishes the six term list types (total, partial injective, partial non-injective, each combined with order or unorder) and

| function | term | CIQLOG expression |
|---|---|---|
| | $\mathrm{tq}\langle \mathit{query} \rangle$ | $= \mathrm{tq}(\varnothing)\langle \mathit{query} \rangle$ |
| | $\mathrm{tq}(\mathcal{E})\langle \mathtt{and}(t_1, t_2) \rangle$ | $= (\mathcal{E}_2, (Q_1 \wedge Q_2))$ **where** $(\mathcal{E}_1, Q) = \mathrm{tq}(\mathcal{E})\langle t_1 \rangle \qquad (\mathcal{E}_2, Q) = \mathrm{tq}(\mathcal{E}_1)\langle t_2 \rangle$ |
| | $\mathrm{tq}(\mathcal{E})\langle \mathtt{or}(t_1, t_2) \rangle$ | $= (\mathcal{E}_2, (Q_1 \vee Q_2))$ **where** $(\mathcal{E}_1, Q) = \mathrm{tq}(\mathcal{E})\langle t_1 \rangle \qquad (\mathcal{E}_2, Q) = \mathrm{tq}(\mathcal{E}_1)\langle t_2 \rangle$ |
| | $\mathrm{tq}(\mathcal{E})\langle \mathtt{not}(t) \rangle$ | $= (\mathcal{E}', \neg(Q))$ **where** $(\mathcal{E}', Q) = \mathrm{tq}(\mathcal{E})\langle t \rangle$ |
| | $\mathrm{tq}(\mathcal{E})\langle \mathit{qterm} \rangle$ | $= (\mathcal{E}', \mathtt{root}(r) \wedge Q)$ **where** $(\mathcal{E}', r, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}, \bot, ())\langle \mathit{qterm} \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathit{tid} \mathbin{@} t \rangle$ | $= (\mathcal{E}', \{v\}, Q)$ **where** $v = \mathcal{E}(\mathit{tid})$ if defined otherwise, if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E} +\!\!+ (\mathit{tid}, v), v, \mathcal{V})\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{idvar}\ X \mathbin{@} t \rangle$ | $= (\mathcal{E}'', \{v\}, E \wedge Q)$ **where** $v = \mathcal{E}(\mathit{tid})$ if defined otherwise, if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', E) = \mathrm{tq}_{\mathit{var}}(\mathcal{E}, v, \doteq)\langle X \rangle$<br>$(\mathcal{E}'', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}', v, \mathcal{V})\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{position}\ \mathit{num} \mathbin{@} t \rangle$ | $= (\mathcal{E}', \{v\}, @^L(v, \mathit{num}) \wedge Q$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}, v, \mathcal{V})\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{position\ var}\ X \mathbin{@} t \rangle$ | $= (\mathcal{E}'', \{v\}, E \wedge Q$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', E) = \mathrm{tq}_{\mathit{var}}(\mathcal{E}, v, =_{@})\langle X \rangle$<br>$(\mathcal{E}'', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}', v, \mathcal{V})\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{var}\ X \to t \rangle$ | $= (\mathcal{E}''', \{v\}, E' \wedge Q$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', E) = \mathrm{tq}_{\mathit{var}}(\mathcal{E}, v, \triangleq)\langle X \rangle$<br>if $\mathrm{isLabel}(\mathtt{X}) \in \mathcal{E}$, $E' = E \wedge \mathrm{outdeg}(v, \mathtt{0})$ otherwise<br>$E' = E$ and $\mathcal{E}'' = \mathcal{E}' +\!\!+ \{\mathrm{isTerm}(\mathtt{X}, v)\}$<br>$(\mathcal{E}''', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}'', \mathcal{V})\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{desc}\ t \rangle$ | $= (\mathcal{E}'', \{v\}, F_{\mathit{struct}} \wedge Q$ **where** if $v = \bot$, $v$ new variable<br>$F_{\mathit{struct}} = \mathrm{tq}_{\mathit{struct}}(v, \mathcal{V})$<br>$(\mathcal{E}'', V, Q) = \mathrm{tq}_{\mathit{term}}(\mathcal{E}, \bot, (v, \varnothing, \varnothing, \mathrm{path}^L_*, \top, \top))\langle t \rangle$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathit{label}\ \mathit{children} \rangle$ | $= (\mathcal{E}_2, v, F_1 \wedge F_2 \wedge F_{\mathit{struct}})$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}_1, F_1) = \mathrm{tq}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{label} \rangle$<br>$(\mathcal{E}_2, F_2) = \mathrm{tq}_{\mathit{child}}(\mathcal{E}_1, v)\langle \mathit{children} \rangle$<br>$F_{\mathit{struct}} = \mathrm{tq}_{\mathit{struct}}(v, \mathcal{V})$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \text{"}\mathit{string}\text{"} \rangle$ | $= (\mathcal{E}', v, F \wedge F_{\mathit{struct}})$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', F) = \mathrm{tq}_{\mathit{label}}(\mathcal{E}, v)\langle \mathit{string} \rangle$<br>$F_{\mathit{struct}} = \mathrm{tq}_{\mathit{struct}}(v, \mathcal{V})$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \hat{}\mathit{tid} \rangle$ | $= (\mathcal{E} +\!\!+ (\mathit{tid}, v), v, F_{\mathit{struct}})$ **where** $v = \mathcal{E}(\mathit{tid})$ if defined, otherwise $v$ new variable<br>$F_{\mathit{struct}} = \mathrm{tq}_{\mathit{struct}}(v, \mathcal{V})$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{var}\ X \rangle$ | $= (\mathcal{E}', v, E \wedge F_{\mathit{struct}})$ **where** if $v = \bot$, $v$ new variable<br>$(\mathcal{E}', E) = \mathrm{tq}_{\mathit{var}}(\mathcal{E}, v, \triangleq)\langle X \rangle$<br>$F_{\mathit{struct}} = \mathrm{tq}_{\mathit{struct}}(v, \mathcal{V})$ |
| $\mathrm{tq}_{\mathit{term}}$ | $(\mathcal{E}, v, \mathcal{V})\langle \mathtt{without}\ \mathit{tlist} \rangle$ | $= \mathrm{tq}_{\mathit{tlist}}(\mathcal{E}, \mathcal{V})\langle \mathit{tlist} \rangle$ |

Table 20. Translating Xcerpt query terms: queries and query terms

calls for each $\mathsf{tq}_{tlist}$ with different parameters. The relation to the parent variable is always $\mathsf{path}^L$ where $L = \{\textsc{child}, \textsc{comment}, \textsc{value}\}$ as stated above. But between the siblings, the relations vary: In the unordered, partial, non-injective case, there are no such relations at all (indicated by $\top$), in the case of partial, but injective terms we add complemented identity join, in the case of total terms we add a limitation on the out-degree of the parent (case 3). When combined with order, we obtain transitive-reflexive sibling order for partial, non-injective, transitive sibling order for partial, injective, and, again, an additional out-degree constraint for the total case (cases 4–6). **(5)** The actual formula of the relations between the variables is defined in $\mathsf{tq}_{tlist}$ which is called whenever translating lists of query terms (i.e., in $\mathsf{tq}_{child}$ and in the without case of $\mathsf{tq}_{term}$). $\mathsf{tq}_{tlist}$ does not merely translate the given lists in the given order, but delays the translation of negative sub-terms until all positive ones are translated. This is necessary to allow the relations between negative and positive sub-terms (even ones after the negative sub-term) to be contained in the scope of the $\neg$. Otherwise, we demand the existence of such a relation rather than demand its non-existence. Nevertheless we ensure that all relations are enforced by gathering the CIQLog variables returned by the translation of all positive sub-terms to the right of a negative one in that sub-terms $V_i^{right}$.

To illustrate $\mathsf{tq}_{tlist}$ consider the Xcept query term

```
a[[ b, without( c, without( d ) ), without( e ), f ]]
```

It is necessary to ensure that in the resulting CIQLog expression all references to a CIQLog variable for the translation of, e.g., c are within a negation. This includes sibling relations to, e.g., the translation of f. Otherwise we would, falsely, require that there has to be a (following) sibling relation to f-labeled node, instead of requiring that there is no c with such a relation.

Returning to $\mathsf{tq}_{term}$ in Table 20, we see that all the "prefixes" of a query term (case 1–6) are translated along the same scheme: if necessary, we take a new variable and add some relations on that variable to the output of the translation of the contained query term. For the identifier case (case 1) we only modify the term environment. Identity variables (case 2), position variables (case 4), and term restrictions (case 5) are translated using appropriate $\doteq$, $=_@$, and $\triangleq$, respectively, between the involved CIQLog variables. Descendant sub-terms are a bit like the translation of a singleton term list using $\mathsf{tq}_{tlist}$: we do not need to establish any relations to left or right siblings (as there are none) but can directly translate the single child, but using the transitive closure $\mathsf{path}_*^L$ instead of $\mathsf{path}^L$ as relation between parent and child variable. The translation of the basic terms (case 7–10) is fairly straightforward: we translated label parts, if there are any, then children, if there are any, and finally establish structural relations between

| function term | CIQLog expression |
|---|---|
| $\mathrm{tq}_{tlist}(\mathcal{E},\mathcal{V})\langle t_1,\ldots,t_n\rangle$ | $= (\mathcal{E}_n, \bigcup V_i^+, \bigwedge_{t_i \text{ positive}} F_i \wedge \neg\left(\bigwedge_{t_i \text{ negative}} F_i\right)$<br>**where** $\mathcal{E}_0 = \mathcal{E}$, $V_0 = \mathcal{V}.V_{\leftarrow}$<br>$t_i$ positive: $V_i^+ = V_i$, $V_i^- = \varnothing$, $V_i^{left} = \bigcup_{j<i} V_j^+$, $V_i^{right} = \mathcal{V}.V_{\rightarrow}$<br>$t_i$ negative: $V_i^+ = \varnothing$, $V_i^- = V_i$, $V_i^{left} = \bigcup_{j<i} V_j$, $V_i^{right} = \bigcup_{j>i} V_j^+ \cup \mathcal{V}.V_{\rightarrow}$<br>$(\mathcal{E}_i, V_i, F_i) = \mathrm{tq}_{term}(\mathcal{E}_{i-1}, \bot, (\mathcal{V}.v_{\uparrow}, V_i^{left}, V_i^{right}, \mathcal{V}.r_{\uparrow}, \mathcal{V}.r_{\leftarrow}, \mathcal{V}.r_{\rightarrow}))$ |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle\{\{\{\ tlist\}\}\}\rangle$ | $= (\mathcal{E}, F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \top, \top))\langle tlist\rangle$ |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle\{\{\ tlist\}\}\rangle$ | $= (\mathcal{E}, F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \dddot{=}, \dddot{=}))\langle tlist\rangle$ |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle\{\ tlist\}\rangle$ | $= (\mathcal{E}, \mathrm{outdeg}^L(v_{\uparrow}, n) \wedge F)$<br>**where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \dddot{=}, \dddot{=}))\langle tlist\rangle$<br>$n$ number of terms in *tlist* |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle[[[\ tlist]]]\rangle$ | $= (\mathcal{E}, \mathfrak{D}^L(v_p) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \ll_*^L, \ll_*^L))\langle tlist\rangle$ |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle[[\ tlist]]\rangle$ | $= (\mathcal{E}, \mathfrak{D}^L(v_p) \wedge F)$ **where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \ll_+^L, \ll_+^L))\langle tlist\rangle$ |
| $\mathrm{tq}_{child}(\mathcal{E},v_{\uparrow})\langle[\ tlist]\rangle$ | $= (\mathcal{E}, \mathrm{outdeg}^L(v_{\uparrow}, n) \wedge \mathfrak{D}^L(v_p) \wedge F)$<br>**where** $(\mathcal{E}, V, F) = \mathrm{tq}_{tlist}(\mathcal{E}, (v_{\uparrow}, \varnothing, \varnothing, \mathrm{path}^L, \ll_+^L, \ll_+^L))\langle tlist\rangle$<br>$n$ number of terms in *tlist* |
| $\mathrm{tq}_{label}(\mathcal{E},v)\langle *\rangle$ | $= (\mathcal{E}, \top)$ |
| $\mathrm{tq}_{label}(\mathcal{E},v)\langle label\rangle$ | $= (\mathcal{E}, \mathfrak{L}(v, label))$ |
| $\mathrm{tq}_{label}(\mathcal{E},v)\langle \mathtt{var\ X}\rangle$ | $= \begin{cases} (\mathcal{E} +\!\!+ \mathrm{isLabel}(\mathrm{X}), \cong (v, \mathcal{E}(\mathrm{X}) \wedge F)) & \text{if } \exists v' : (\mathrm{X}, v') \in \mathcal{E} \text{ and } F = \bigwedge_{\mathrm{isTerm}(\mathrm{X},v)\in\mathcal{E}} \mathrm{outdeg}(v, 0) \\ (\mathcal{E} +\!\!+ (\mathrm{X}, v) +\!\!+ \mathrm{isLabel}(\mathrm{X}), \top) & \text{otherwise} \end{cases}$ |
| $\mathrm{tq}_{var}(\mathcal{E},v,\mathrm{eq})\langle \mathrm{X}\rangle$ | $= \begin{cases} (\mathcal{E}, v \cong \mathcal{E}(\mathrm{X})) & \text{if } \exists v' : \mathrm{isLabel}(\mathrm{X}) \in \mathcal{E} \wedge (\mathrm{X}, v') \in \mathcal{E} \\ (\mathcal{E}, v \text{ eq } \mathcal{E}(\mathrm{X})) & \text{if } \exists v' : \mathrm{isLabel}(\mathrm{X}) \notin \mathcal{E} \wedge (\mathrm{X}, v') \in \mathcal{E} \\ (\mathcal{E} +\!\!+ (\mathrm{X}, v), \top) & \text{otherwise} \end{cases}$ |
| $\mathrm{tq}_{struct}(v_{curr},\mathcal{V})$ | $= r_{\uparrow}(v_{\uparrow}, v_{curr}) \wedge \bigwedge_{v_l \in V_{\leftarrow}} r_{\leftarrow}(v_l, v_{curr}) \wedge \bigwedge_{v_r \in V_{\rightarrow}} r_{\rightarrow}(v_r, v_{curr})$ |
| $\mathrm{tq}_{struct}(v_{curr},\mathcal{V})$ | $= r_{\uparrow}(v_{\uparrow}, v_{curr})$ if $r_{\rightarrow} = r_{\leftarrow} = \top$ |

Table 22. Translating Xcerpt query terms: term lists, variables, and labels

the parent and sibling variables using $\mathsf{tq}_{struct}$.

EXAMPLES. To illustrate the translation of query terms, we once again turn back to the examples from Section 7.3, in this case the example query terms in Tables 9 and 11. Tables 24 and 26 show the translation to CIQLog for some of these query terms as well as the environment passed to the translation of a corresponding rule head. If these query terms occur as top-level query terms, we have to add a root relation on the top-level CIQLog variable that we omit in the following for space reasons.

| | Query term | | cIQLog expression | $\mathcal{E}$ |
|---|---|---|---|---|
| T1 | `a{ }` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 0)$ | $\varnothing$ |
| T2 | `a[ ]` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 0) \wedge \mathfrak{O}^L(v_1)$ | $\varnothing$ |
| T3 | `a{ b }` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 1) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b)$ | $\varnothing$ |
| T4 | `a{ b, b }` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 2) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, b) \wedge v_2 \doteq v_3$ | $\varnothing$ |
| P1 | `a{{ b }}` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b)$ | $\varnothing$ |
| P2 | `a[[ b, c ]]` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, c) \wedge v_2 \ll_+ v_3$ | $\varnothing$ |
| I1 | `a{{{ b, b }}}` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, b)$ | $\varnothing$ |
| I2 | `a[[[ b, b, d ]]]` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, b) \wedge \text{CHILD}(v_1, v_4) \wedge \mathfrak{L}(v_4, d) \wedge v_2 \ll_* v_3 \wedge v_2 \ll_* v_4 \wedge v_3 \ll_* v_4$ | $\varnothing$ |
| D1 | `a{ desc b }` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 1) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}_*(v_2, v_3) \wedge \mathfrak{L}(v_3, b)$ | $\varnothing$ |
| D2 | `a{ desc b, desc c }` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{outdeg}^L(v_1, 2) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_1, v_3) \wedge v_2 \doteq v_3 \wedge \text{CHILD}_*(v_2, v_4) \wedge \mathfrak{L}(v_4, b) \wedge \text{CHILD}_*(v_3, v_5) \wedge \mathfrak{L}(v_5, c)$ | $\varnothing$ |
| D3 | `a{{ desc b, desc c }}` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_1, v_3) \wedge v_2 \doteq v_3 \wedge \text{CHILD}_*(v_2, v_4) \wedge \mathfrak{L}(v_4, b) \wedge \text{CHILD}_*(v_3, v_5) \wedge \mathfrak{L}(v_5, c)$ | $\varnothing$ |
| D4 | `a{{{ desc b, desc c }}}` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{CHILD}(v_1, v_2) \wedge \text{CHILD}(v_1, v_3) \wedge \text{CHILD}_*(v_2, v_4) \wedge \mathfrak{L}(v_4, b) \wedge \text{CHILD}_*(v_3, v_5) \wedge \mathfrak{L}(v_5, c)$ | $\varnothing$ |
| W1 | `a{{ b, without( c ), d }}` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, d) \wedge v_2 \mathrel{C\doteq} v_3 \wedge \neg\big(\text{CHILD}(v_1, v_4) \wedge \mathfrak{L}(v_4, c) \wedge v_2 \mathrel{C\doteq} v_4 \wedge v_3 \mathrel{C\doteq} v_4\big)$ | $\varnothing$ |
| W2 | `a[[ b, without( c ), d ]]` | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1, a) \wedge \mathfrak{O}^L(v_1) \wedge \text{CHILD}(v_1, v_2) \wedge \mathfrak{L}(v_2, b) \wedge \text{CHILD}(v_1, v_3) \wedge \mathfrak{L}(v_3, d) \wedge v_2 \ll_+ v_3 \wedge \neg\big(\text{CHILD}(v_1, v_4) \wedge \mathfrak{L}(v_4, c) \wedge v_2 \ll_+ v_4 \wedge v_4 \ll_+ v_3\big)$ | $\varnothing$ |

Notice in Table 24 in particular the subtle, but essential differences in the translations for total (T1–T2), partial, but injective (P1–P2), and partial and non-injective (I1–I2) query terms. I2 is an example where more than necessary sibling relations are generated. This is the case for all order relations between siblings as the translation above does not exploit their transitivity. This can be easily recognized and removed in a post-processing step. Alternatively, one can adapt the translation to handle ordered and unordered term lists differently. D1, D4 show that when translating desc we always generate an intermediate child step to express sibling relations on that child step. However, if there are no such relations (because it is the only sub-term or because we are in a partial, non-injective term) we can avoid the child step and use $\text{path}_+^L$ instead of $\text{path}^L$ followed by $\text{path}_*^L$. Again this is a general equivalence for CIQLOG queries on data graph relations and can be optimized in a post-processing.

| | Query term | | CIQLOG expression | $\mathcal{E}$ |
|---|---|---|---|---|
| V1 | a{ var X } | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,1) \wedge \text{CHILD}(v_1,v_2)$ | $\{(X,v_2)\}$ |
| V2 | a{{ var X }} | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2)$ | $\{(X,v_2)\}$ |
| V3 | a{{ var X, var X }} | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \text{CHILD}(v_1,v_3) \wedge v_2\ \mathsf{C} \doteq v_3 \wedge v_2 \triangleq v_3$ | $\{(X,v_2)\}$ |
| V4 | a{{{ var X, var X }}} | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \text{CHILD}(v_1,v_3) \wedge v_2 \triangleq v_3$ | $\{(X,v_2)\}$ |
| V5 | a{ var X{ var X } } | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \text{CHILD}(v_2,v_3) \wedge \text{outdeg}^L(v_3,0) \wedge v_2 \cong v_3$ | $\{(X,v_2), \text{isLabel}(X)\}$ |
| V6 | a{ var X → c, var X } | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \mathfrak{L}(v_2, c) \wedge \text{CHILD}(v_1,v_3) \wedge v_2 \triangleq v_3$ | $\{(X,v_2)\}$ |
| V7 | a{ desc var X } | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{outdeg}^L(v_1,1) \wedge \text{CHILD}(v_1,v_2) \wedge \text{CHILD}_*(v_2,v_3)$ | $\{(X,v_3)\}$ |
| V8 | a{{var X, without(var X)}} | $\xrightarrow{\text{tq}_{term}}$ | $\mathfrak{L}(v_1,a) \wedge \text{CHILD}(v_1,v_2) \wedge \neg\big(\text{CHILD}(v_1,v_3) \wedge v_2\ \mathsf{C} \doteq v_3 \wedge v_2 \triangleq v_3\big)$ | $\{(X,v_2)\}$ |

Table 26. Query terms containing variables and their CIQLOG translation

Query terms with variables are considered in Table 26. Notice, in par-

ticular V3 and V6 that illustrate multiple occurrences of the same variable in a total or partial, injective query term: here we demand that matches for the two terms are not the same node (i.e., in the complement relation of $\doteq$) but have the same label and structure (i.e., stand in $\triangleq$ relation). In V5, the effect of a label occurrence of a variable is illustrated.

It is easy to see from the translation functions that the following result holds (considering that each case covers at least one Xcerpt construct and no case introduce duplication of the translation of its sub-expressions).

**Theorem 7.1.** *The size of the* CIQLOG *expression $Q$ returned by $tr_{Xcerpt}$ for a given Xcerpt rule $P$ is linear in the size of $P$.*

## 7.5 FROM NON-RECURSIVE, SINGLE-RULE CORE XCERPT TO FULL XCERPT

In the previous sections, we focus on Xcerpt$^{core,NR,SR}$, i.e., the non-recursive, single-rule fragment of Xcerpt. Notice, however, that the above translation generates a CIQLOG rule from a Core Xcerpt rule and is applicable whether there are only one or many rules in a translated Xcerpt program.

Thus, for translating full Core Xcerpt, i.e., possibly recursive, multi-rule Xcerpt but with the restrictions from Section 7.2 wrt. each rule, we can use the above translation unchanged. Some improvements of the resulting CIQLOG program can be achieved by exploiting that Xcerpt programs are grouping and negation stratified. Before the translation, we select one such stratification and rewrite the program to introduce root terms with unique labels for each stratum. This allows for easier recognition of said strata after the translation to CIQLOG, where we only have to consider the root variable and its label to limit rule chaining to each stratum.

Moving from Core Xcerpt to full Xcerpt is less obvious. Aside from numerous, but essentially easy specificities such as namespaces, attributes, comments, processing-instructions etc. there are a few constructs that merit a closer consideration. Among them are:

**(1)** `optional` TERMS: Optional QUERY terms can be rewritten to combinations of `without` and `or` but require specific treatment in heads.[2] For that CIQLOG provides conditional construction which precisely addresses optional construct terms.

**(2)** `order-by` CLAUSES: `order-by` clauses in Xcerpt allow for different lists of order variables than grouping variables whereas Xcerpt$^{core,NR,SR}$ always assumes that both lists are identical. This is supported by

2 This is the case, as we can not always split the Xcerpt rule in two rules on the Xcerpt level due to grouping expressions.

CIQLog (the variable lists in order do not have to coincide with the ones in new) and can be easily added to the translation by a new $\mathcal{E}$.order sequence of variables.

**(3)** GROUPING OVER TERM LISTS instead of single terms yields results not expressible in Xcerpt$^{\text{core,NR,SR}}$: `a[b, all(var X, var Y), d]` yields

```
a[b, x1, y1, x2, y2, ..., xn, yn, d]
```

where $x_i$ is the binding for X in the $i$-th binding tuple, i.e., bindings for X and Y are paired wrt. term order. This can be accommodated in the translation by nesting order expressions as in the translation of XQuery (cf. Chapter 8).

**(4)** ADDITIONAL GROUPING EXPRESSIONS such as some and first groupings are to some extent expressible using aggregation operators in CIQLog. However, in general, some is not expressible in CIQLog as its result is non-deterministic and CIQLog expresses only deterministic queries (up to isomorphisms on invented values).

**(5)** CONDITIONS IN XCERPT QUERY TERMS can be translated to CIQLog if appropriate relations or functions are available (or added) to CIQLog.

# 8 TRANSLATING XQUERY



## 8.1 INTRODUCTION

Among XML query languages, XPath and, increasingly, XQuery play the dominant role to such an extent that motivating their use in this work has become superfluous. XPath's properties, evaluation, complexity, containment, etc. have been studied extensively in recent years, for a survey see [22]. For XQuery, most research still focuses on implementation and evaluation aspects, e.g., [93, 155, 39]. In the following, we present a novel semantics for XPath and XQuery by translation to CIQLog that also serves as foundation for the evaluation of XPath and XQuery with the CIQCAG algebra defined in Part IV. Together with the CIQCAG algebra, we achieve the first implementation of XQuery that scales from tree queries to graph queries, from tree data to graph data. Its time and space complexity is as good as or better than the complexities of previous systems limited to, e.g., tree queries on tree data (for details on the complexity see Part IV).

The translation of XPath and XQuery to CIQLog is the focus of this chapter. For the most part, we use navigational XPath (introduced in [113]) and

composition-free Core XQuery[1] (introduced in [144]), two important and convenient fragments of XPath and XQuery that can be translated to non-recursive CIQLog. The extension to full XQuery is only briefly outlined in Section 8.4. The essential limitation of both fragments compared to the full languages is that all relations in the query are only on nodes of the input tree but not on nodes by the query itself. This limitation is similar to the limitation to Xcerpt$^{\text{core,NR,SR}}$ in Chapter 7.

In addition to providing a path for implementing XPath and XQuery using CIQCAG, the translation also gives a purely logical semantics for both languages where previous semantics for XQuery are functional or algebraic (see also Table 31). This sheds new light on some of the differences between composition-free XQuery on the one hand and XPath on the other hand, in particular, on the effect of nested **for** loops and element construction in XQuery.

## 8.2 TRANSLATING XPATH

*Introducing XPath*

Unary selection of XML elements is, by now, almost always done using XPath or some variant of XPath (such as XPointer). XPath provides an elegant and compact way of describing "paths" in an XML document (represented as in to Chapter 5.3 but without resolution of ID/IDREF links). Paths are made up of "steps" each specifying a direction, called *axis*, in which to navigate through the document, e.g., child, following, or ancestor, cf. Figure 38 for the full set of axes. Together with the axis, a step contains a restriction on the type or label of the data items to be selected, called *node test*. Node tests may be labels of element or attribute nodes, node kind wildcards such as * (any node with some label), element(), node(), text(), or comment(). Any step may be adorned by one or more *qualifiers* each expressing additional restrictions on the selected nodes. Compared to languages such as XQuery, Xcerpt, or even SPARQL, the most distinctive feature of XPath is the lack of explicit variables. This makes it impossible to express $n$-ary queries and limits XPath, for the most part, to two-variable logic [153, 36].

*XPath examples*

For instance, the XPath expression `/descendant::paper/child::author` consists of two steps, the first selecting `paper` elements that are descendants of the root ("of the root" is indicated by the leading slash), the second selecting `author` children of such `paper` elements. More interesting queries can be expressed by exploiting XPath's qualifiers, e.g., the following XPath expression that selects all authors that are also PC members of a conference

1 In the following, we omit the "Core" where no confusion is possible.

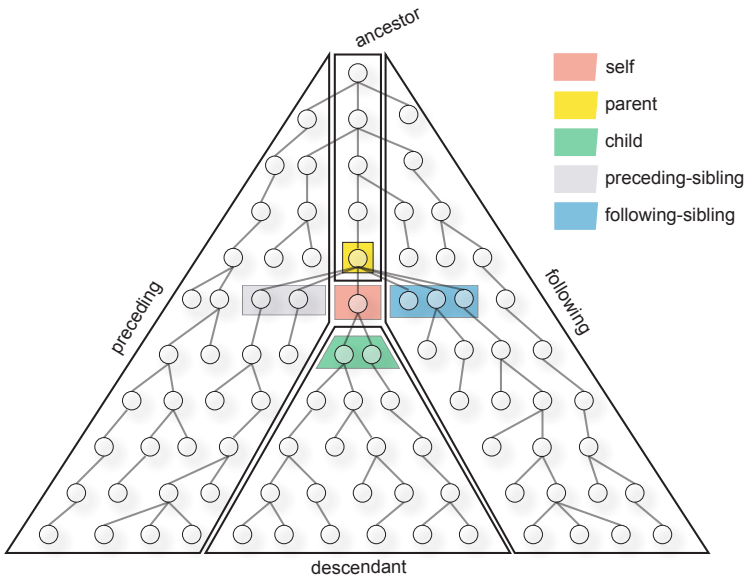



Figure 38. XPath axis (from [103])

(more precisely that have node children with the same label):

```
/child::conference/descendant::paper/child::author[child::node()
    =
 /child::conference/child::member/child::node()]
```

In addition to the strict axis plus node test notation, XPath uses also an abbreviated syntax where child axis may be omitted, descendant is (roughly) abbreviated by // etc. In the following, we only use the full syntax. We also limit ourselves to the core feature of XPath as discussed here and thus present a view of XPath similar to Navigational XPath of [112] and [22]. Due to [170], we also limit ourselves to forward axes such as child and following, rewriting expressions with reverse axes such as parent, ancestor, or preceding where necessary.

### 8.2.1 SYNTAX AND SEMANTICS

*XML tree as relational structure*

Following [22], we define the semantics of XPath over a relational structure as introduced in Section 5.6.1: An XML-tree is considered a relational structure $T$ over the schema $((\mathsf{Lab}^\lambda)_{\lambda\in\Sigma}, R_{\text{child}}, R_{\text{nest-sibling}}, relRoot)$. The nodes of this tree are labeled using the symbols from $\sigma$ which are queried using $\mathfrak{Q}^\lambda$ (note, that $\lambda$ is a single label not a label set as in the graph relations of Chapter 5). The parent-child relations are represented by $R_{\text{child}}$. The order between siblings is represented by $R_{\text{nest-sibling}}$. The root node of the tree is identified by root. It is easy to see that this view of XML trees (which is as in [22] or [144]), makes an XML-tree a specific instance of a CIQLog data graph, cf. Chapter 5. There are some additional derived relations, viz. $R_{\text{descendant}}$, the transitive, $R_{\text{descendant-or-self}}$ the transitive reflexive closure of $R_{\text{child}}$, $R_{\text{following-sibling}}$, the transitive closure of $R_{\text{next-sibling}}$, $R_{\text{self}}$ relating each node to itself, and $R_{\text{following}}$ the composition of $R^{-1}_{\text{descendant-or-self}} \circ R_{\text{following-sibling}} \circ R_{\text{descendant-or-self}}$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label.

*Syntax of navigational XPath*

The syntax of navigational XPath is defined as follows (again following [112] and [22]):

$$
\begin{array}{lll}
\langle path\rangle & ::= & \langle step\rangle \mid \langle step\rangle \text{ '/' } \langle path\rangle \mid \langle path\rangle \text{ '}\cup\text{'} \langle path\rangle \mid \text{'/' } \langle path\rangle \\
\langle step\rangle & ::= & \langle axis\rangle \text{ '::' } \langle node\text{-}test\rangle \mid \langle step\rangle \text{'['} \langle qualifier\rangle \text{']'} \\
\langle axis\rangle & ::= & \texttt{'child'} \mid \texttt{'descendant'} \mid \texttt{'descendant-or-self'} \\
 & \mid & \texttt{'next-sibling'} \mid \texttt{'following-sibling'} \mid \texttt{'following'} \\
\langle node\text{-}test\rangle & ::= & \langle label\rangle \mid \texttt{'node()'} \\
\langle qualifier\rangle & ::= & \langle path\rangle \mid \langle path\rangle \text{ '}\wedge\text{'} \langle path\rangle \mid \langle path\rangle \text{ '}\vee\text{'} \langle path\rangle \mid \text{'}\neg\text{'} \langle path\rangle \\
 & \mid & \texttt{'lab()'} \text{ '='} \lambda\text{'} \\
 & \mid & \langle path\rangle \text{ '='} \langle path\rangle
\end{array}
$$

$$
\begin{array}{ll}
[\![ axis ]\!]_{\text{Nodes}}(n) & = \{(n' : R_{axis}(n, n')\} \\
[\![ \lambda ]\!]_{\text{Nodes}}(n) & = \{(n' : \text{Lab}^{\lambda}(n')\} \\
[\![ \texttt{node()} ]\!]_{\text{Nodes}}(n) & = \text{Nodes}(T) \\
[\![ axis\texttt{::}nt\texttt{[}qual\texttt{]} ]\!]_{\text{Nodes}}(n) & = \{n' : n' \in [\![ axis ]\!]_{\text{Nodes}} \wedge n' \in [\![ nt ]\!]_{\text{Nodes}} \wedge [\![ qual ]\!]_{\text{Bool}}(n')\} \\
[\![ step/path ]\!]_{\text{Nodes}}(n) & = \{n'' : n' \in [\![ step ]\!]_{\text{Nodes}}(n) \wedge n'' \in [\![ path ]\!]_{\text{Nodes}}(n')\} \\
[\![ path_1 \cup path_2 ]\!]_{\text{Nodes}}(n) & = [\![ path_1 ]\!]_{\text{Nodes}}(n) \cup [\![ path_2 ]\!]_{\text{Nodes}}(n) \\
\hline
[\![ path ]\!]_{\text{Bool}}(n) & = [\![ path ]\!]_{\text{Nodes}}(n) \neq \varnothing \\
[\![ path_1 \wedge path_2 ]\!]_{\text{Bool}}(n) & = [\![ path_1 ]\!]_{\text{Bool}}(n) \wedge [\![ path_2 ]\!]_{\text{Bool}}(n) \\
[\![ path_1 \vee path_2 ]\!]_{\text{Bool}}(n) & = [\![ path_1 ]\!]_{\text{Bool}}(n) \vee [\![ path_2 ]\!]_{\text{Bool}}(n) \\
[\![ \neg path ]\!]_{\text{Bool}}(n) & = \neg [\![ path ]\!]_{\text{Bool}}(n) \\
[\![ \texttt{lab()} = \lambda ]\!]_{\text{Bool}}(n) & = \text{Lab}^{\lambda}(n) \\
[\![ path_1 = path_2 ]\!]_{\text{Bool}}(n) & = \exists n', n'' : n' \in [\![ path_1 ]\!]_{\text{Nodes}}(n) \wedge n'' \in [\![ path_2 ]\!]_{\text{Nodes}}(n) \\
 & \quad \wedge \cong (n', n'')
\end{array}
$$

Table 27. Semantics for navigational XPath (following [22])

*Semantics of navigational XPath*

The semantics of a navigational XPath expression over a relational structure $T$ representing an XML tree (as defined above) is defined in Table 27 by means of $[\![\ ]\!]_{\text{Nodes}}(n)$ where $n$ is a node, called *context node*. $[\![\ ]\!]_{\text{Nodes}}(n)$ associates each XPath expression and context node with a set of nodes that constitutes the semantics of that expression if evaluated with the given context node. It uses $[\![\ ]\!]_{\text{Bool}}(n)$ for the semantics of qualifiers under a context node $n$.

For details on the semantics as well as differences to full XPath see [22].

### 8.2.2 TRANSLATION

*Translation examples*

Consider again the above examples. The first (`/descendant::paper/child::author`) is translated to the following CIQLOG rule:

```
ans(v3) ⟵ root(v1) ∧ CHILD+(v1,v2) ∧ 𝔏(v2,paper) ∧
  CHILD(v2,v3) ∧ 𝔏(v3,author)
```

We use ans as the canonical answer predicate containing the (single) answer variable whose bindings represent the results of an XPath expression. Just as the original expression, the body of the CIQLOG rule selects descendants of the root with label paper and of those the author children. The

latter are propagated to the head of the rule.

The second example is as follows

```
/child::conference/descendant::paper/child::author[child::node()
    =
 /child::conference/child::member/child::node()]
```

and translated similarly but illustrates qualifiers and nested queries:

```
ans(v4) ⟵ root(v1) ∧ CHILD(v1,v2) ∧ 𝔏(v2,conference) ∧
  CHILD+(v2,v3) ∧ 𝔏(v3,paper) ∧ CHILD(v3,v4) ∧ 𝔏(v4,author) ∧
  CHILD(v4,v5) ∧ ≅(v5,w4) ∧ root(w1) ∧ CHILD(w1,w2) ∧
  𝔏(w2,conference) ∧ CHILD(w2,w3) ∧ 𝔏(w3, member) ∧ CHILD(w3,w4)
```

*Translation function*

In general, we translate a navigational XPath expression using the $\mathrm{tr}_{\mathrm{XPath}}$ specified in Table 29. $\mathrm{tr}_{\mathrm{XPath}}$ returns a CIQLog formula that realizes the given XPath expression as well as an answer variable. We wrap $(Q, w) = \mathrm{tr}_{\mathrm{XPath}}(\bot, \bot)$ where $\bot$ is an arbitrary variable into a CIQLog rule $\mathtt{ans}(w) \longleftarrow Q$.

The translation in Table 29 is fairly straightforward. In a slight abuse of notation, we allow the direct use $\mathrm{tr}_{\mathrm{XPath}}$ in a formula. In this case we ignore the second return value (i.e., the answer variable). The translation is parametrized by the parent variable $v$ and the current variable $v'$. Axis are mapped to corresponding relations using the helper function $\mathrm{relation}(\mathit{axis})$), as are label node-steps (cases 1–3). $\mathrm{relation}(\mathit{axis})$ is defined in the obvious way:

$$\begin{aligned}
\mathrm{relation}(\text{child}) &= \textsc{child} & \mathrm{relation}(\text{descendant}) &= \textsc{child}_+ \\
\mathrm{relation}(\text{descendant-or-self}) &= \textsc{child}_* & \mathrm{relation}(\text{next-sibling}) &= \ll^{\textsc{child}} \\
\mathrm{relation}(\text{following-sibling}) &= \ll^{\textsc{child}}_+ & \mathrm{relation}(\text{following}) &= \blacktriangleleft^{\textsc{child}}_+
\end{aligned}$$

Steps are translated by translating axis, node-test, and qualifier separately. For the translation of the qualifier a new variable is created and the qualifier is translated with the old current variable as parent and the new variable as current variable. Notice, how we ignore the answer variable returned by the qualifier. This small difference to the translation of a path outside a qualifier implements the existential semantics of XPath qualifiers. Together with the path operator / (cases 5–6), the qualifier is the only context-changing expression where we move from one query variable to the next one. The inner path operator (case 5) translates the leftmost step and then continues with the translation of the remainder of the path using a new current variable. Absolute XPath expressions (expressions starting with /) are translated in case 7 where we use *fresh* variables rather than $v$ and $v'$. Thus, the only possible link between nested absolute expressions is the answer variable (which is used, e.g., for unions or joins). In unions,

| function | XPath expression | CIQLog expression |
|---|---|---|
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle axis\rangle$ | | $= (\mathrm{relation}(axis)(v, v'), v')$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle \lambda\rangle$ | | $= (\mathfrak{L}(v', \lambda), v')$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle \mathtt{node()}\rangle$ | | $= (\top, v')$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle axis\mathtt{::}node\text{-}test[qualifier]\rangle$ | | $= (\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle axis\rangle \wedge \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle node\text{-}test\rangle \wedge \mathrm{tr}_{\mathrm{XPath}}(v', v'')\langle qualifier\rangle, v')$ **where** $v''$ is a new variable |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle step/path\rangle$ | | $= (\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle step\rangle \wedge F, w)$ **where** $v''$ is a new variable $(F, w) = \mathrm{tr}_{\mathrm{XPath}}(v', v'')\langle path\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle /path\rangle$ | | $= (\mathrm{root}(v'') \wedge F, w)$ **where** $v'', v'''$ are new variables $(F, w) = \mathrm{tr}_{\mathrm{XPath}}(v'', v''')\langle path\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1 \cup path_2\rangle$ | | $= ((F_1 \vee F_2) \wedge (w \doteq w_1 \vee w \doteq w_1), w)$ **where** $w$ is a new variable $(F_1, w_1) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1\rangle \quad (F_2, w_2) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_2\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1 \vee path_2\rangle$ | | $= ((F_1 \vee F_2), w_1)$ **where** $(F_1, w_1) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1\rangle$ $(F_2, w_2) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_2\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1 \wedge path_2\rangle$ | | $= (F_1 \wedge F_2, w_1)$ **where** $(F_1, w_1) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1\rangle$ $(F_2, w_2) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_2\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle \neg path\rangle$ | | $= (\neg(F), w)$ **where** $(F, w) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle \neg path\rangle$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle \mathtt{lab()} = \lambda\rangle$ | | $= (\mathfrak{L}(v, \lambda), v)$ |
| $\mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1 = path_2\rangle$ | | $= (F_1 \wedge F_2 \wedge w_1 \cong w_2, w_1)$ **where** $(F_1, w_1) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_1\rangle \quad (F_2, w_2) = \mathrm{tr}_{\mathrm{XPath}}(v, v')\langle path_2\rangle$ |

Table 29. Translating navigational XPath

conjunctions, disjunctions, and joins (cases 7–9, 12) we translate each operand separately but combine the result differently, most notably for conjunction, disjunction, and join (cases 8, 9, 12) we do not care about the answer variable (they only occur in qualifiers where, as described above, we drop answer variables anyway). Finally, label equality (case 11) is translated just like a labeled node test (case 2), but on the parent variable instead of the current variable due to the "context" switch of the qualifier.

**Theorem 8.1.** *Let $P$ be a navigational XPath expression and $(Q, w) = tr_{XPath}(\bot, \bot)\langle P\rangle$. Then the relation* ans *defined by* ans$(w) \longleftarrow Q$ *is $[\![ P ]\!]_{Nodes}$ and the size of $Q$ is linear in the size of $P$.*

*Proof (Sketch).* First, consider only path expression without qualifiers. It is easy to see that, given appropriate base relations, the resulting CIQLog expressions express the same query as the XPath expressions. Note, that the answer variable is by translation the variable for the last step, as in XPath. With qualifiers, the same observation holds (as answer variables returned by paths in qualifiers are ignored by definition, see case 4). Note that the join in case 12 is a label join (as in the semantics of navigational XPath). In full XPath = is a join on the string value of a node, which is a notion covered neither in navigational XPath as defined in [22] nor by the CIQLog translation.

It is easy to see from Table 29, that the resulting CIQLog expression: each case translates at least one construct in $P$ and in no case is the result of a sub-translation duplicated. □

## 8.3 FROM XPATH TO COMPOSITION-FREE XQUERY

Turning from XPath to a larger fragment of XQuery that is amenable to translation into CIQLog, we choose non-compositional XQuery as defined in [144] (though we follow more closely the variant in [21]).

### 8.3.1 COMPOSITION-FREE XQUERY IN 1000 WORDS

Though not nearly as common as XPath, XQuery has nevertheless achieved the status of predominant XML query language, at least as far as database products and research are concerned (in total, XSLT [72] is probably still more widely supported and used). XQuery is essentially an extension of XPath (though some of its axis are only optional in XQuery), but most of XPath becomes syntactic sugar in XQuery. This is particularly true for XPath qualifiers which can be reduced to **`where`** or **`if`** clauses in XQuery. Indeed, the XQuery standard is accompanied [89] by a normalization of XQuery to a core dialect of the language.

*Introducing composition-free*

Here, we consider first an important, if somewhat academic fragment of XQuery, viz. composition-free XQuery as defined in [144] and [21]. It is slightly academic as we restrict the syntax far more than necessary to minimize the constructs to be consider for the formal semantics of composition-free XQuery as well as for the translation to CIQLog. However, many of the restrictions to the syntax can be dropped (e.g., we could integrate full navigational XPath as discussed in Section 8.2) without affecting expressiveness and complexity, see also [21]. The only real restriction of composition-free XQuery in comparison to full XQuery is that it disallows any querying of constructed nodes, i.e., the domain of all relations is limited to the input nodes. This limitation clearly does not hold for full XQuery (even if we do not consider user-defined functions) and its effect on expressiveness and complexity is discussed in detail in [144].

*Example*

(Composition-free) XQuery is built around controlled iterations over nodes of the input tree, expressed using **for** expressions. Controlled iteration is important for XQuery as it founded on sequences of nodes rather than sets of nodes (as XPath 1.0 and CIQLog). In this respect it is more similar to languages such as DAPLEX [192] or OQL [67] than to XPath or Xcerpt. (**For**) loops use XPath expressions for navigation and XML-look-a-likes for element construction all of which can be, essentially, freely nested. The following query gives an example of XQuery expressions. It creates a paperlist containing one author element for each author in the input XML tree (bound here and in the following to the canonical input variable `$inp`). For each such author, the nested **for** loop creates a list of all its papers. The latter expression can be more elegantly expressed in full XQuery using XPath qualifiers or **where** clauses but here it is shown in the "normalized" syntax of composition-free XQuery after [144].

```
<paperlist>
 for $a in $inp/descendant::author return
   <author> for $p in $inp/descendant::paper return
     if some $x in $p/child::author satisfies deep-equal($x, $a)
     then $p
   </author>
</paperlist>
```

### 8.3.2 SYNTAX

A full definition of the syntax of composition-free XQuery as used here is given in Table 30. It deviates only marginally from [144] and [21]. In addition to the specification in Table 30, the usual semantic restrictions apply, e.g., the label of the start and end tags must be the same, variables

| | | |
|---|---|---|
| ⟨*query*⟩ | ::= | ⟨*query*⟩ ⟨*query*⟩ \| ⟨*element*⟩ \| ⟨*variable*⟩ |
| | \| | ⟨*step*⟩ \| ⟨*iteration*⟩ \| ⟨*conditional*⟩ |
| ⟨*element*⟩ | ::= | '<' ⟨*label*⟩ '>' ⟨*query*⟩ '<' ⟨*/label*⟩ '>' |
| | \| | '<' 'lab(' ⟨*variable*⟩ ')>' ⟨*query*⟩ '</' 'lab(' ⟨*variable*⟩ ')>' |
| ⟨*step*⟩ | ::= | ⟨*variable*⟩ '/' ⟨*axis*⟩ '::' ⟨*node-test*⟩ |
| ⟨*iteration*⟩ | ::= | 'for' ⟨*variable*⟩ 'in' ⟨*step*⟩ 'return' ⟨*query*⟩ |
| ⟨*conditional*⟩ | ::= | 'if' ⟨*condition*⟩ 'then' ⟨*query*⟩ |
| ⟨*condition*⟩ | ::= | ⟨*variable*⟩ '=' ⟨*variable*⟩ \| ⟨*variable*⟩ '=' '<' ⟨*label*⟩ '/>' \| 'true' |
| | \| | 'some' ⟨*variable*⟩ 'in' ⟨*step*⟩ 'satisfies' ⟨*condition*⟩ |
| | \| | ⟨*condition*⟩ 'and' ⟨*condition*⟩ \| ⟨*condition*⟩ 'or' ⟨*condition*⟩ \| 'not' ⟨*condition*⟩ |
| ⟨*axis*⟩ | ::= | 'child' \| 'descendant' \| 'descendant-or-self' |
| | \| | 'next-sibling' \| 'following-sibling' \| 'following' |
| ⟨*node-test*⟩ | ::= | ⟨*label*⟩ \| 'node()' |
| ⟨*variable*⟩ | ::= | '$'⟨*identifier*⟩ |

Table 30. Syntax of composition-free XQuery

must be defined (using **for**) before use, etc. As stated, there is one exception from the latter, viz. the canonical input variable `$inp` which is always bound to the input XML tree.

*Three forms of equality*

In Table 30, we use a general equality. XQuery provides in fact three kinds of equality, viz. node, atomic (or value), and deep equality which correspond roughly to $\doteq$, $\cong$, and $\overset{\circ}{=}_{bij}$ of CIQLog data graphs. For all forms of equality the productions of Table 30 apply.

*Restrictions*

Again, compared to full XQuery the principle omission is the ability to query constructed nodes or values. In the syntax, this leads most prominently to the restriction of expressions following **in** in a **for**, i.e., expressions that provide bindings for variables, to XPath steps with variables. This way variables are always bound only to nodes from the input tree (anything reachable from `$inp` using XPath expressions). Another important omission is the absence of **let** clauses, which provide set-valued variables to XQuery. Conditional expressions are normalized to **if** clauses, where XQuery offers XPath qualifiers, **where** clauses, and **if** clauses.

Though **order-by** clauses are omitted, the result of an XQuery expres-

sion is always an ordered tree and the order of node construction must be precisely preserved (as given by the iteration of the **for** clauses which iterated over their respective node sequences mostly in document order).

### 8.3.3 SEMANTICS

*XML trees as relational structures*

The formal semantics for composition-free XQuery is, for the most part, closely aligned with the one for XPath discussed above. Again we considered an XML tree a relational structure $T$ over the schema $((\mathsf{Lab}^{\lambda})_{\lambda\in\Sigma}, R_{\mathsf{child}}, R_{\mathsf{nest\text{-}sibling}}, \mathsf{root})$. The nodes of this tree are labeled using the symbols from $\sigma$ which are queried using $\mathfrak{L}^{\lambda}$ (note, that $\lambda$ is a single label not a label set as in the graph relations of Chapter 5). The parent-child relations are represented by $R_{\mathsf{child}}$. The order between siblings is represented by $R_{\mathsf{nest\text{-}sibling}}$. The root node of the tree is identified by root. It is easy to see that this view of XML trees (which is as in [22] or [144]), makes an XML-tree a specific instance of a ClQLog data graph, cf. Chapter 5. There are some additional derived relations, viz. $R_{\mathsf{descendant}}$, the transitive, $R_{\mathsf{descendant\text{-}or\text{-}self}}$ the transitive reflexive closure of $R_{\mathsf{child}}$, $R_{\mathsf{following\text{-}sibling}}$, the transitive closure of $R_{\mathsf{next\text{-}sibling}}$, $R_{\mathsf{self}}$ relating each node to itself, and $R_{\mathsf{following}}$ the composition of $R^{-1}_{\mathsf{descendant\text{-}or\text{-}self}} \circ R_{\mathsf{following\text{-}sibling}} \circ R_{\mathsf{descendant\text{-}or\text{-}self}}$. Finally, we can compare nodes based on their label using $\cong$ which contains all pairs of nodes with same label. In addition to the XPath relations, XQuery also considers two more forms of equality: one based on node identity, $=_{\mathsf{nodes}}$ which relates each node to itself, and deep equality $=_{\mathsf{deep}}$ which holds for two nodes if there exists an isomorphism between their respective sub-trees.

For example, the XML document $\langle a\rangle_1\ \langle b/\rangle_2\ \langle c\rangle_3\langle c/\rangle_4\langle /c\rangle\ \langle /a\rangle$ (denoting node id's by integer subscripts) is represented as $T = (\mathsf{Lab}^{a} = \{1\}, \mathsf{Lab}^{b} = \{2\}, \mathsf{Lab}^{c} = \{3,4\}, R_{\mathsf{child}} = \{(1,2),(1,3),(3,4)\}, R_{\mathsf{nest\text{-}sibling}} = \{(2,3)\}, \mathsf{root} = \{1\})$ over the label alphabet $\{a,b,c\}$. All other relations can be derived from this definition, see also Chapter 6.

*Semantics of composition-free XQuery*

In the following, we also allow unions of such structures, i.e., XML "forests". The semantics of a composition-free XQuery expression is then defined, following [21], using $[\![\ ]\!]$ over a given such forest and a list of nodes from that forest $\vec{e} = [e_1, \ldots, e_n]$ that represent bindings for variables $x_1, \ldots, x_n$. For that, we assume that all variables are first renamed to $x_i$ such that $i$ is the number of variables in whose scope $x_i$ is declared and assuming that $inp is scoped over the entire query. E.g., the query

```
for $x in $inp/child::a return
  for $y in $x/child::b return $x
for $z in $inp/child::c return
  for $v in $inp/child::d return $v
```

becomes

```
for $2 in $1/child::a return
  for $3 in $2/child::b return $2
for $2 in $1/child::c return
  for $3 in $1/child::d return $3
```

In the following, we assume that queries are in the latter form.

Table 31 specifies the semantics of composition-free XQuery on an XML forest $F$ and a binding vector $\vec{e} = [e_1, \ldots, e_n]$ which is initially of length 1 containing bindings for `$inp`, i.e., usually one (or more, if querying XML collections) root node(s).

The semantics uses three auxiliary notions. **(1)** $\uplus$ is the union on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \uplus (F_2, \vec{e}_2) = (F_1 \cup F_2, \vec{e}_1 \circ \vec{e}_2)$ where $\circ$ is list (or vector) concatenation and the union of XML forests is defined component by component. **(2)** $\owedge$ is the intersection on pairs of XML forests and binding vectors such that $(F_1, \vec{e}_1) \owedge (F_2, \vec{e}_2) = (F_1, [e_i \in \vec{e}_1 : e_i \in \vec{e}_2])$. Note, that we only preserve $F_1$ (and thus $\owedge$ is *not* associative). However, for the purpose of the semantics the choice of the XML forest is arbitrary as $\owedge$ is only used for the semantics of conditions for which only the existence or non existence (and not their actual value) of bindings is relevant for the semantics of the full query. **(3)** $\mathsf{construct}(l, (F, [w_1, \ldots, w_n]))$ denotes construction of a new tree where $l$ is a label, $F$ is an XML forest and $[w_1, \ldots, w_n]$ is a vector of nodes in $F$. It returns a pair $(F \cup T', [\mathsf{root}(T')])$ where $T'$ is a tree over a new set of nodes whose root $\mathsf{root}(T')$ is labeled with $l$ and with the $i$-th subtree of $\mathsf{root}(T')$ isomorphic to the sub-tree rooted at $w_i$ in $F$. Furthermore construct is assumed to return a tree with a distinct set of nodes each time it is called. This corresponds to *value invention* in CIQLog.

Using these definitions, the semantics is fairly straightforward. In [21], Benedikt and Koch point out that most of the condition expressions (cases 10, 12–16) can be reduced to other XQuery expressions and thus do not need to be addressed in the semantics. We choose to give their definitions directly as the resulting expressions are no longer in composition-free XQuery.

*Explaining the semantics*

The crucial parts of the semantics are cases 2 and 3, that illustrate element construction, case 7 that illustrates iteration, and case 8, the semantics of conditionals. The other cases are very similar to XPath and mostly just return appropriate binding vectors but leave $F$ unchanged. Element construction (case 2 and 3) is achieved using the aforementioned construct function and returns a forest containing the newly constructed tree and bindings pointing to that tree's root node. Iteration using **for** has almost exactly the same semantics as the path separator / in XPath: the **return**

$$
\begin{aligned}
&[\![\ \texttt{()}\ ]\!]\,(F,\vec{e}) && = (F,[\,]) \\
&[\![\ \texttt{<}l\texttt{>}\ q\ \texttt{</}l\texttt{>}\ ]\!]\,(F,\vec{e}) && = \mathsf{construct}(l,[\![\ q\ ]\!]\,(F,\vec{e})) \\
&[\![\ \texttt{<lab(\$}x_i\texttt{)>}\ q\ \texttt{</lab(\$}x_i\texttt{>}\ ]\!]\,(F,[e_1,\ldots,e_n]) \\
& && = \mathsf{construct}(\texttt{lab}(e_i),[\![\ q\ ]\!]\,(F,[e_1,\ldots,e_n])) \\
&[\![\ \texttt{\$}x_i\ ]\!]\,(F,[e_1,\ldots,e_n]) && = (F,[e_i]) \\
&[\![\ \texttt{\$}x_i\texttt{/}\mathit{axis}\texttt{::}l\ ]\!]\,(F,[e_1,\ldots,e_n]) && = (F,[d : R_{\mathit{axis}}(e_i,d) \wedge \mathsf{Lab}^l(d)] \\
&\left[\!\!\left[\ q_1\ q_2\ \right]\!\!\right](F,\vec{e}) && = \left[\!\!\left[\ \mathit{query}_1\ \right]\!\!\right](F,\vec{e}) \uplus \left[\!\!\left[\ \mathit{query}_1\ \right]\!\!\right](F,\vec{e}) \\
&[\![\ \texttt{for \$}x_i\ \texttt{in}\ s\ \texttt{return}\ q\ ]\!]\,(F,\vec{e}) && = \biguplus_{l\in\vec{l}} [\![\ q\ ]\!]\,(F,\vec{e}\cdot l) \text{ where } (F,\vec{l}) = [\![\ s\ ]\!]\,(F,\vec{e}) \\
&[\![\ \texttt{if}\ \mathit{cond}\ \texttt{then}\ q\ ]\!]\,(F,\vec{e}) && = \begin{cases} [\![\ \mathit{query}\ ]\!]\,(F,\vec{e}) & \text{if } \pi_2([\![\ \mathit{cond}\ ]\!]\,(F,\vec{e})) \neq [\,] \\ (F,[\,]) & \text{otherwise} \end{cases} \\
&[\![\ \texttt{\$}x_i\texttt{/}\mathit{axis}\texttt{::node()}\ ]\!]\,(F,[e_1,\ldots,e_n]) && = (F,[d : R_{\mathit{axis}}(e_i,d)]) \\
&[\![\ \texttt{some \$}x_i\ \texttt{in}\ s\ \texttt{satisfies}\ c\ ]\!]\,(F,\vec{e}) && = [\![\ \texttt{for \$}x_i\ \texttt{in}\ s\ \texttt{return}\ c\ ]\!]\,(F,\vec{e}) \\
&\left[\!\!\left[\ \texttt{\$}x_i = \texttt{\$}x_j\ \right]\!\!\right](F,[e_1,\ldots,e_n]) && = \begin{cases} (F,[e_i]) & \text{if } e_i = e_j \\ (F,[\,]) & \text{otherwise} \end{cases} \\
&[\![\ \texttt{\$}x_i = \texttt{<}l\texttt{/>}\ ]\!]\,(F,[e_1,\ldots,e_n]) && = \begin{cases} (F,[e_i]) & \text{if = atomic equal and } \mathsf{Lab}^l(e_i) \\ (F,[e_i]) & \text{if = deep equal, } \mathsf{Lab}^l(e_i), \\ & \text{and } \nexists\ d : R_{\mathsf{child}}(e_i,d) \\ (F,[\,]) & \text{otherwise} \end{cases} \\
&[\![\ c_1\ \texttt{or}\ c_2\ ]\!]\,(F,\vec{e}) && = [\![\ c_1\ ]\!]\,(F,\vec{e}) \uplus [\![\ c_2\ ]\!]\,(F,\vec{e}) \\
&[\![\ c_1\ \texttt{and}\ c_2\ ]\!]\,(F,\vec{e}) && = [\![\ c_1\ ]\!]\,(F,\vec{e}) \pitchfork [\![\ c_2\ ]\!]\,(F,\vec{e}) \\
&[\![\ \texttt{not}\ c\ ]\!]\,(F,\vec{e}) && = \begin{cases} (F,[\mathsf{root}(F)]) & \text{if } (F',[\,]) = [\![\ c\ ]\!]\,(F,\vec{E}) \\ (F,[\,]) & \text{otherwise} \end{cases} \\
&[\![\ \texttt{true}\ ]\!]\,(F,\vec{e}) && = (F,[\mathsf{root}(F)])
\end{aligned}
$$

Table 31. Semantics for composition-free XQuery (following [21])

expression is evaluated in the context of the **in** part, just like the subordinate path is evaluated in the context of the superordinate one. Indeed, the XQuery normalization transforms path expressions consisting of multiple steps to **for** loops as in composition-free XQuery. The difference is, of course, that the semantics of the **return** may be nodes from a newly constructed tree. It is crucial that this is the case *only* for the semantics of the **return** expression, not for that of the **in** expression which never modifies the given XML forest. In full XQuery, this does not hold, the **in** is followed by an arbitrary expression. Finally, conditionals are (again reminiscent of qualifiers in XPath) translated using a non-empty test on the bindings returned by the condition.

Note that the *relations of the input forest are never changed*. We may add new forests, but those do not have any relations to the input forest.

It is worth noting, that the semantics is uniform for boolean-valued conditions and for node-valued expressions (in contrast to the XPath case in Section 8.2). This follows [21] and allows a more compact definition of the semantics, at the cost of slightly surprising definitions for boolean operations and `true` in the latter part of the semantics. In the translation, we separate boolean-valued conditions from other expressions by a separate translation function as in the XPath case.

### 8.3.4 TRANSLATION

For the translation of composition-free XQuery, the main challenge lies in the "constructive" part of composition-free XQuery not in the selection part. In fact, compared to XPath, the selection part is enriched by only three significant features: the ability to use variables and thus to refer back to previously established bindings, the presence of deep-equal, and the ability to sequence expressions and thus their results. Moreover, the latter is the only feature that is challenging for the translation as it requires somewhat more sophisticated management of sibling order than in the translation for XPath. In a sense, this is already part of the constructive part, i.e., element construction and the translation of the results of queries contained in element construction. This is what ends up in the head of a CIQLog rule (which in the XPath case above is always a single atom over the unary answer variable).

*Example*

Consider again the XQuery example from above:

```
<paperlist>
  for $a in $inp/descendant::author return
    <author> for $p in $inp/descendant::paper return
      if some $x in $p/child::author satisfies deep-equal($x, $a)
      then $p
```

```
  </author>
</paperlist>
```

What is the result of this query, if there is no author in the document? What if there is an author with no paper? In XPath, if any part of a path (disregarding **or** for the moment) has no match the entire query has no match. In XQuery, this is not the case. The above example *always* yields at least a paperlist element. It may be empty, if there are no authors in the document but it may never be absent. The some for the inner loop: The author element is constructed and included in the result for any author in the input even if there is no paper for an author. Of course, we change this behavior by placing additional **if** clauses. But in general, an XQuery expression may always return an empty set of nodes, but never causes other expressions not contained in it to fail.

TRANSLATION EXAMPLE. Continuing with this example, how can we express the same query as a CIQLog (in fact, $\text{CIQLog}^{\text{NR}}$) rule? The following CIQLog rule shows the answer to that question:

$$
\begin{aligned}
&\texttt{root}(\mathsf{id}_1(i)) \wedge \mathfrak{D}(\mathsf{id}_1(i),\textsc{child}) \wedge \\
&\quad \textsc{child}(\mathsf{id}_1(i),\mathsf{id}_2(i),\underline{\texttt{order}}(\top,0)) \wedge \mathfrak{L}(\mathsf{id}_2(i),\texttt{paperlist}) \wedge \mathfrak{D} \\
&\qquad (\mathsf{id}_2(i),\textsc{child}) \wedge \\
&\quad \textbf{if } v_1 \neq \textbf{nil} \textbf{ then } \big( \\
&\qquad \textsc{child}(\mathsf{id}_2(i),\mathsf{id}_3(i,v_1),\underline{\texttt{order}}(\underline{\texttt{order}}(\top,0),0,v_1)) \wedge \mathfrak{L}(\mathsf{id}_3(i,v_1), \\
&\qquad\quad \texttt{author}) \wedge \\
&\qquad \textbf{if } v_2 \neq \textbf{nil} \textbf{ then } ( \\
&\qquad\quad \textbf{if } v_3 \neq \textbf{nil} \textbf{ then } ( \\
&\qquad\qquad \textsc{child}(\mathsf{id}_3(i,v_1),\ \mathsf{id}_4(i,v_1,v_2),\underline{\texttt{order}}(\underline{\texttt{order}}(\top,0),0,v_2)) \wedge \\
&\qquad\qquad \texttt{deep-copy}(\mathsf{id}_4(i,v_1,v_2),v_2)\ )\ )\ \big) \\
&\longleftarrow \texttt{root}(i) \wedge \big(\textsc{child}_+(i,\ v_1) \wedge \mathfrak{L}(v_1,\ \texttt{author}) \wedge \\
&\qquad (\textsc{child}_+(i,\ v_2) \wedge \mathfrak{L}(v_2,\ \texttt{paper}) \wedge \\
&\qquad\quad (\textsc{child}(v_2,\ v_3) \wedge \mathfrak{L}(v_3,\ \texttt{author}) \wedge v_3 \stackrel{\circ}{=}_{bij} v_1 \\
&\qquad\qquad \vee\ v_3 = \textbf{nil}) \\
&\qquad\quad \vee\ v_2 = \textbf{nil}) \\
&\qquad \vee\ v_1 = \textbf{nil}\big)
\end{aligned}
$$

*Abbreviating construction*

The same abbreviations of head formulas as in the translation of Xcerpt in Chapter 7 are used:

$$\textsc{child}(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_2(\vec{x}_2),o)$$

abstracts the edge construction necessary in CIQLog and thus is an abbreviation for

$$\circ\!\!\rightarrow(\mathsf{id}_1(\vec{x}_1),\mathsf{id}_N(\vec{x}_2)) \wedge \rightarrow\!\!\circ(\mathsf{id}_2(\vec{x}_2),\mathsf{id}_N(\vec{x}_2)) \wedge \texttt{pos}(\mathsf{id}_N(\vec{x}_2),\underline{\texttt{order}}(\top,0))$$

We also implicitly omit the document order $\prec$ as parameter for all order terms. That is, two binding vectors for the same query variables are ordered by looking at the bindings of the query variables in sequence and considering their relative position in document order $\prec$ (which is provided, e.g., by $\text{CHILD}_+(n, n') \vee \blacktriangleleft_+^{\text{CHILD}}(n, n')$, i.e., $n$ is before $n'$ if it is a ancestor of $n'$ or before $n'$ in $\blacktriangleleft_+^{\text{CHILD}}$.

*Conditional construction*

The most notable difference to the translations for XPath or even Xcerpt is the extraordinary amount of conditional construction (and corresponding $\vee v = \textbf{nil}$ for some variable $v$). As discussed above, this is due to the nature of XQuery expressions where non matching sub-expression often do not affect the matching of their superordinate expressions. Each of the conditionals "guards" one sub-expression and ensures that it is omitted from the result if there are no bindings for the guard variable (but without affecting the remainder of the head construction).

*Order terms*

The other striking feature of the translation are the unusually (and at first glance, unnecessarily) complex order terms (e.g., line 4 and line 7). They are necessary to allow arbitrary occurrences of **for** loops (and thus arbitrarily long sequences of constructed elements) to occur in arbitrary positions in sequences of XQuery expressions within the same element constructor. First, notice, that the order terms in line 4 and 7 are the same. This is possibly since order is only relevant between edges with the same source and these order terms are on edges with different source. In the translation below this is reflected by "resetting" the nesting of order terms at any element construction (cases 2, 3 in Table 34).

*Adding outside the outer loop*

To further illustrate the need for nested order terms, let's add another XQuery expression before the outer **for** loop but within the paperlist element:

```
1 <paperlist>
    <abc>()</abc>
3   for $a in $inp/descendant::author return
      <author> for $p in $inp/descendant::paper return
5       if some $x in $p/child::author satisfies deep-equal($x, $a)
        then $p
7     </author>
  </paperlist>
```

The body of the resulting CIQLog rule is unchanged, but in the head we have to construct the new element, but also to adapt the order terms (omitting the unchanged body):

$$\text{root}(\text{id}_1(i)) \wedge \mathfrak{O}(\text{id}_1(i), \text{CHILD}) \wedge$$
$$\text{CHILD}(\text{id}_1(i), \text{id}_2(i), \underline{\text{order}}(\top, 0)) \wedge \mathfrak{L}(\text{id}_2(i), \text{paperlist}) \wedge \mathfrak{O}(\text{id}_2(i), \text{CHILD}) \wedge$$
$$\text{CHILD}(\text{id}_2(i), \text{id}_n(i), \underline{\text{order}}(\top, 0)) \wedge \mathfrak{L}(\text{id}_n(i), \text{abc}) \wedge \mathfrak{O}(\text{id}_n(i), \text{CHILD}) \wedge$$

```
4  if v1 ≠ nil then (
     CHILD(id2(i),id3(i,v1),order(order(⊤,1),0,v1)) ∧ 𝔏(id3(i,v1),
         author) ∧
6    if v2 ≠ nil then (
       if v3 ≠ nil then (
8        CHILD(id3(i,v1), id4(i,v1,v2),order(order(⊤,0),0,v2)) ∧
         deep-copy(id4(i,v1,v2),v2) ) ) )
```

In line 5, we no longer use offset 0 but offset 1. That is $t_1 = \mathsf{order}(\top, 0)$, the order term for the new abc, is always smaller (wrt. the order on order invention terms defined in Section 6.2.1) than $t_2 = \mathsf{order}(\mathsf{order}(\top, 1), 0, v_1))$ regardless of the binding for $v_1$. Here, that is the case as the parent order term of $t_2$ is the same as $t_1$ up to the offset which is higher and thus $t_1 < t_2$.

Similar changes occur if the added element is at the end or a **for** loop (in which case the parent order terms of the elements created by the original and by the new **for** loop are the same, except for the offset).

*Adding inside the outer loop*

Instead of adding abc before the loop, we might also want to add it within the loop:

```
1 <paperlist>
   for $a in $inp/descendant::author return
3    <abc>()</abc>
     <author> for $p in $inp/descendant::paper return
5     if some $x in $p/child::author satisfies deep-equal($x, $a)
      then $p
7    </author>
  </paperlist>
```

Again, this can be addressed by adapting the order terms, in this case the new element shares the same order term as the author element contained in the loop, but with a smaller offset:

```
root(id1(i)) ∧ 𝔇(id1(i),CHILD) ∧
2  CHILD(id1(i),id2(i),order(⊤,0)) ∧ 𝔏(id2(i),paperlist) ∧ 𝔇
       (id2(i),CHILD) ∧
   if v1 ≠ nil then (
4    CHILD(id2(i),idn(i,v1),order(order(⊤,0),0,v1)) ∧ 𝔏(idn(i,v1),abc) ∧
       𝔇(idn(i,v1),CHILD) ∧
6    CHILD(id2(i),id3(i,v1),order(order(⊤,0),1,v1)) ∧ 𝔏(id3(i,v1),
         author) ∧
     if v2 ≠ nil then (
8      if v3 ≠ nil then (
         CHILD(id3(i,v1), id4(i,v1,v2),order(order(⊤,0),0,v2)) ∧
10       deep-copy(id4(i,v1,v2),v2) ) ) )
```

*Eliminating intermediary element construction*

The nesting level of order terms increases only if **for** loops are contained within each other without intermediate element construction as in the

final example (where we delete the author around the inner loop).

```
<paperlist>
 for $a in $inp/descendant::author return
   <abc>()</abc>
   for $p in $inp/descendant::paper return
    if some $x in $p/child::author satisfies deep-equal($x, $a)
    then $p
</paperlist>
```

Now the order terms for the elements created by the inner loop depend on the order terms for the outer loop. It is still ensured, that the order term for the abc is before the order terms of all the elements in the inner loop since it is the same as their parent order term except for the offset which is smaller.

```
root(id1(i)) ∧ 𝔇(id1(i),CHILD) ∧
 CHILD(id1(i),id2(i),order(⊤,0)) ∧ 𝔏(id2(i),paperlist) ∧ 𝔇
      (id2(i),CHILD) ∧
 if v1 ≠ nil then (
  CHILD(id2(i),idn(i,v1),order(order(⊤,0),0,v1)) ∧ 𝔏
       (idn(i,v1),abc) ∧ 𝔇(idn(i,v1),CHILD) ∧
  if v2 ≠ nil then (
    if v3 ≠ nil then (
      CHILD(id2(i), id3(i,v1,v2)
           ,order(order(order(⊤,0),1,v1),0,v2)) ∧
      deep-copy(id3(i,v1,v2),v2) ) ) )
```

TRANSLATION FUNCTION. The translation of composition-free XQuery expressions to CIQLOG is specified by $\mathrm{tr}_{\mathrm{XQuery}}$. As for the translation of Xcerpt, we use an environment $\mathcal{E}$ containing mappings from XQuery variables to CIQLOG variables and the list of current iteration variables. In addition, we use $\mathcal{O}$ to hold information about the current order term. $\mathcal{O}$ is a triple $\langle g : \mathit{term}, o : \mathit{offset}, i : \mathit{order\ vars}\rangle$ where $g$ is the current parent order term, $o$ the current offset, and $i$ the list of order variables. We use $\mathcal{O}.g$, $\mathcal{O}.o$, and $\mathcal{O}.i$ to refer to each of the components. If there is no parent order term, we use the canonical empty order term $\top$ from Section 6.2.1. Finally, we write $\mathsf{order}(\mathcal{O})$ as abbreviation for $\mathsf{order}(\mathcal{O}.g, \mathcal{O}.o, \mathcal{O}.i)$.

Given an XQuery expression $P$, $\mathrm{tr}_{\mathrm{XQuery}}$ computes a corresponding CIQLOG expression as follows:

$$\mathrm{tr}_{\mathrm{XQuery}}\langle \mathsf{P}\rangle = \mathsf{root}(r_c) \wedge \mathfrak{D}^{\mathrm{CHILD}}(r_c) \wedge C \longleftarrow \mathsf{root}(r_q) \wedge Q$$
$$\textbf{where } r_q \text{ be a new variable, } r_c = \mathsf{id}(r_q) \text{ with id a new identifier}$$
$$(C, Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}, (\top, \mathsf{o}, []), r_c)\langle P\rangle \text{ with}$$
$$(\mathtt{inp}, r_q) \in \mathcal{E} \text{ and } \mathcal{E}.\mathsf{iter} = [r_q].$$

As for Xcerpt, this is mainly a wrapper adding root restrictions and root construction around the result of the actual translation which is computed by $\mathrm{tr}_{\mathrm{XQuery}}$ with a new environment initialized to contain a mapping for the canonical XQuery input variable `$inp` to the root of the XML document and an iteration sequence containing only the CIQLOG variable corresponding to `$inp`. Thus, for each root node of the input document (of which there is exactly one, if the input is an XML tree, but may be several if we allow XML forests) a (distinct) result for $P$ is computed. The order environment $\mathcal{O}$ is initialized with $(\top, \mathrm{o}, [])$, i.e., the empty order term, offset o, and no iteration variables. Finally, we also pass $r_c$ the parent *construct variable* to $\mathrm{tr}_{\mathrm{XQuery}}$

| function | XQuery | CIQLOG expression |
|---|---|---|
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle()\rangle$ | | $= (\top, \top)$ |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`<l>` $q$ `</l>`$\rangle$ | | $= (\mathfrak{L}(v,l) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})) \wedge \mathfrak{D}^{\mathrm{CHILD}}(v) \wedge C, Q)$<br>**where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier<br>$(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}, (\top, \mathrm{o}, []), p)\langle q\rangle$ |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`<lab($`$x_i$`)>` $q$ `</lab($`$x_i$`)>`$\rangle$ | | $= (\cong (\mathcal{E}(x_i), v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})) \wedge \mathfrak{D}^{\mathrm{CHILD}}(v) \wedge C, Q)$<br>**where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier<br>$(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}, (\top, \mathrm{o}, []), p)\langle q\rangle$ |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`$`$x_i$$\rangle$ | | $= (\text{deep-copy}(\mathcal{E}(x_i), v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})), \top)$<br>**where** $v = \mathrm{id}(\mathcal{E}.\mathrm{iter})$ and id a new identifier |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`$`$x_i$`/`*step*$\rangle$ | | $= (\,$**if** $r \neq$ **nil then** $\text{deep-copy}(r,v) \wedge \mathrm{CHILD}(p,v,\mathrm{order}(\mathcal{O})), (Q_s \vee r =$ **nil**$))$<br>**where** $(Q_s, r) = \mathrm{tqc}(\mathcal{E}')\langle$`$`$x_i$`/`*step*$\rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ r$<br>$v = \mathrm{id}(\mathcal{E}.\mathrm{iter} \circ r)$ and id a new identifier |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q_1\ q_2\rangle$ | | $= (C_1 \wedge C_2, Q_1 \wedge Q_2)$ **where** $(C_1, Q_1) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q_1\rangle$<br>$(C_2, Q_2) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}, (\mathcal{O}.g, \mathcal{O}.o + 1, \mathcal{O}.i), p)\langle q_2\rangle$<br>query sequence is left-associative, i.e., $q_1$ no sequence |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`for $`$x_i$ `in` $s$ `return` $q\rangle$ | | $= (\,$**if** $r \neq$ **nil then** $C, (Q_s \wedge Q \vee r =$ **nil**$))$<br>**where** $(Q_s, r) = \mathrm{tqc}(\mathcal{E})\langle s\rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\mathrm{iter} = \mathcal{E}.\mathrm{iter} \circ r$<br>$(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E}' +\!\!+ (x_i, r), (\mathrm{order}(\mathcal{O}), \mathrm{o}, r), p)\langle q\rangle$ |
| $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle$`if` *cond* `then q`$\rangle$ | | $= (\,$**if** $r \neq$ **nil then** $C, (Q_c \wedge Q \vee r =$ **nil**$))$<br>**where** $(Q_c, r) = \mathrm{tqc}(\mathcal{E})\langle$*cond*$\rangle$<br>$(C,Q) = \mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)\langle q\rangle$ |

Table 34. Translating composition-free XQuery

The $\mathrm{tr}_{\mathrm{XQuery}}(\mathcal{E},\mathcal{O},p)$ is specified in full in Table 34. As stated, it takes an

environment $\mathcal{E}$, an order environment $\mathcal{O}$, a parent construct variable, and, of course, the XQuery expression as parameter. It returns a pair $(C, Q)$ of CIQLOG formulas, $C$ a conjunction of head atoms, $Q$ a body formula. The intuitive semantics of $\text{tr}_{\text{XQuery}}$ is that the result of $C$ applied to the bindings provided by $Q$ (i.e., $C \longleftarrow Q$) is isomorphic to the result of the XQuery expression under the given $\mathcal{E}$, $\mathcal{O}$, and parent construct variable (resp. its XQuery counterpart).

*Understanding the translation function*

The translation uses a helper function tqc for the translation of XQuery conditions which is given in Table 36. Before turning to tqc, note the structural similarities between $\text{tr}_{\text{XQuery}}$ and the above semantics for composition-free XQuery (which is due to [144, 21]). In contrast to that semantics, however, the translation has to split up the XQuery expression into a specification of the construction (for the CIQLOG rule head) and a CIQLOG query expressing relations on the nodes of the input tree. This split makes the nature of XQuery expressions, whether they are mainly about querying the input nodes or about creating output, eminently visible in $\text{tr}_{\text{XQuery}}$. E.g., element construction (cases 2–3) does not affect the query (the $Q$ part), but only the construction. On the other hand, all the conditions (in Table 36) used, e.g., for the **in** expression of **for** loops (case 7) only result in query formula and have influence on the construction. Regarding, the order environment $\mathcal{O}$ and its management, it is worth pointing out, that element construction resets that environment (cases 2–3) by using a $(\top, 0, [])$ for the translation of contained expressions. In contrast, iteration (case 7) adds to the nesting depth of order terms of contained expressions by using $(\text{order}(order(\mathcal{O}), o, r)$ as order environment for the translation of nested expressions. Finally, a sequence of queries (case 6) translates the first query with the given order environment, but increases the offset for each following query. For case 6, we assume that query sequence operator is left-associative, i.e., $q_1$ does not consist in a sequence of two other queries.

*Translating conditions*

As stated, conditions are translated using tqc specified in Table 36, which also takes an environment $\mathcal{E}$ in addition to the condition to be translated and returns pairs of CIQLOG body formulas and query variables. The returned query variable identifies a newly created variable, if there is any. It uses the same helper function $\text{relation}(axis)$ as the translation of XPath, see Section 8.2.2.

*Copy semantics*

In one aspect, we deviate slightly from XQuery and the above semantics for composition-free XQuery: we always construct *new* forest whereas XQuery expressions may return simply lists of bindings into the existing data. In other words, we implicitly assume a root element around any XQuery expression to be translated. Without this assumption the result computed by our translation is only correct up to node identity wrt. the

| function | XQuery | CIQLog expression |
|---|---|---|
| tqc$(\mathcal{E})\langle \$x_i/axis::l \rangle$ | | $= (\text{relation}(axis)(\mathcal{E}(x_i), v) \wedge \mathfrak{Q}(v, l), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ and id new identifier |
| tqc$(\mathcal{E})\langle \$x_i/axis::\texttt{node()} \rangle$ | | $= (\text{relation}(axis)(\mathcal{E}(x_i), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ and id a new identifier |
| tqc$(\mathcal{E})\langle \$x_i = \$x_j \rangle$ | | $= \begin{cases} (\mathcal{E}(x_i) \doteq \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is node equal} \\ (\mathcal{E}(x_i) \cong \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is atomic equal} \\ (\mathcal{E}(x_i) \triangleq_{bij} \mathcal{E}(x_j), \mathcal{E}(x_i)) & \text{if = is deep equal} \end{cases}$ |
| tqc$(\mathcal{E})\langle \$x_i = \texttt{<}l\texttt{/>} \rangle$ | | $= \begin{cases} (\texttt{false}, r) & \text{if = is node equal and } r \text{ new var.} \\ (\mathfrak{Q}(\mathcal{E}(x_i), l), \mathcal{E}(x_i)) & \text{if = is atomic equal} \\ (\mathfrak{Q}(\mathcal{E}(x_i), l) \wedge \text{outdeg}(\mathcal{E}(x_i), \text{o}), \mathcal{E}(x_i)) & \text{if = is deep equal} \end{cases}$ |
| tqc$(\mathcal{E})\langle q_1 \text{ and } q_2 \rangle$ | | $= (Q_1 \wedge Q_2 \wedge r = (r_1 \neq \texttt{nil} \wedge r_2 \neq \texttt{nil}), r)$ **where** $r$ is a new variable $(Q_1, r_1) = \text{tqc}(\mathcal{E})\langle q_1 \rangle$ $(Q_2, r_2) = \text{tqc}(\mathcal{E})\langle q_2 \rangle$ |
| tqc$(\mathcal{E})\langle q_1 \text{ or } q_2 \rangle$ | | $= ((Q_1 \vee Q_2) \wedge r = (r_1 \neq \texttt{nil} \vee r_2 \neq \texttt{nil}), r)$ **where** $r$ is a new variable $(Q_1, r_1) = \text{tqc}(\mathcal{E})\langle q_1 \rangle$ $(Q_2, r_2) = \text{tqc}(\mathcal{E})\langle q_2 \rangle$ |
| tqc$(\mathcal{E})\langle \texttt{not } q \rangle$ | | $= (\neg(Q), r)$ **where** $(Q, r) = \text{tqc}(\mathcal{E})\langle q \rangle$ |
| tqc$(\mathcal{E})\langle \texttt{true} \rangle$ | | $= (r = \texttt{true}, r)$ **where** $r$ is a new variable (`true` arbitrary not-**nil** value) |
| tqc$(\mathcal{E})\langle \texttt{some } \$x_i \texttt{ in } s \texttt{ satisfies } c \rangle$ | | $= ((Q_s \wedge Q \vee r = \texttt{nil}), r')$ **where** $(Q_s, r) = \text{tqc}(\mathcal{E})\langle s \rangle$ with $\mathcal{E}' = \mathcal{E}$ but $\mathcal{E}'.\text{iter} = \mathcal{E}.\text{iter} \circ r$ $(Q, r') = \text{tqc}(\mathcal{E}' \mathbin{+\!\!+} (x_i, r))\langle c \rangle$ |

Table 36. Translating composition-free XQuery: conditions

above semantics. This can be addressed by distinguishing between expressions within (at least one) element constructors and those outside of any element constructors. For the latter, we use, instead of deep-copy, direct references to variables and their bindings in the head.

ORDER EXAMPLE. We conclude with a further illustration of the role of order terms in XQuery. Consider the following XQuery program that generates a list of a, b,c, and d tags under a common root r, but the order and number of those tags depends on the number of bindings for $inp/**descendant**::*:

```
<r>
 for $x in $inp/descendant::* return
  <a>()</a>
  for $y in $inp/descendant::* return
   for $z in $inp/descendant::* return
    <b>()</b> <c>()</c>
  <d>()</d>
</r>
```

Assuming, for instance, that $inp/**descendant**::*: matches exactly two nodes in the input, the resulting XML document looks as follows:

```
<r>
 <a />
  <b /> <c /> <b /> <c /> <b /> <c /> <b /> <c />
 <d />
 <a />
  <b /> <c /> <b /> <c /> <b /> <c /> <b /> <c />
 <d />
</r>
```

The query returns as many sequences of a and d elements surrounding sequences of b and c elements as there are matches for `$inp/`**`descendant::*`**.The inner sequence of b's and c's is also repeated once for each match. In CIQLog, we use node invention terms with appropriate grouping variables to obtain as many new nodes as there are matches, and order (invention) terms to ensure that their order is correct. Applying $\mathrm{tr}_{\mathrm{XQuery}}$ to the above program yields:

$$
\begin{aligned}
&\mathtt{root}(\mathsf{id}_1(i)) \wedge \mathfrak{D}(\mathsf{id}_1(i),\textsc{child}) \wedge\\
&\textsc{child}(\mathsf{id}_1(i),\mathsf{id}_2(i),\underline{\mathtt{order}}(\top,0)) \wedge \mathfrak{L}(\mathsf{id}_2(i),\mathtt{r}) \wedge \mathfrak{D}(\mathsf{id}_2(i),\textsc{child}) \wedge\\
&\textbf{if } v_1 \neq \textbf{nil} \textbf{ then } (\\
&\quad \textsc{child}(\mathsf{id}_2(i),\mathsf{id}_3(i,v_1),\underline{\mathtt{order}}(\underline{\mathtt{order}}(\top,0),0,v_1)) \wedge \mathfrak{L}(\mathsf{id}_3(i,v_1),\ \mathtt{a}) \wedge\\
&\quad \textbf{if } v_2 \neq \textbf{nil} \textbf{ then } (\ \textbf{if } v_3 \neq \textbf{nil} \textbf{ then } (\\
&\qquad \textsc{child}\\
&\qquad\quad (\mathsf{id}_2(i),\mathsf{id}_4(i,v_1,v_2,v_3),\underline{\mathtt{order}}(\underline{\mathtt{order}}(\underline{\mathtt{order}}(\top,0),1,v_1),0,v_2,v_3)) \wedge\\
&\qquad \mathfrak{L}(\mathsf{id}_4(i,v_1,v_2),\ \mathtt{b}) \wedge\\
&\qquad \textsc{child}\\
&\qquad\quad (\mathsf{id}_2(i),\mathsf{id}_5(i,v_1,v_2,v_3),\underline{\mathtt{order}}(\underline{\mathtt{order}}(\underline{\mathtt{order}}(\top,0),1,v_1),1,v_2,v_3)) \wedge\\
&\qquad \mathfrak{L}(\mathsf{id}_5(i,v_1,v_2),\ \underline{\mathtt{c}})\ )\ ) \wedge\\
&\quad \textsc{child}(\mathsf{id}_2(i),\mathsf{id}_6(i,v_1),\underline{\mathtt{order}}(\underline{\mathtt{order}}(\top,0),2,v_1)) \wedge \mathfrak{L}(\mathsf{id}_6(i,v_1),\ \mathtt{d})\ )\\
&\longleftarrow\ \mathtt{root}(i) \wedge \big(\textsc{child}_+(i,\ v_1) \wedge\\
&\qquad (\textsc{child}_+(i,\ v_2) \wedge\\
&\qquad\quad (\textsc{child}(v_2,\ v_3) \wedge\\
&\qquad\qquad \vee\ v_3 = \textbf{nil})\\
&\qquad\quad \vee\ v_2 = \textbf{nil})\\
&\qquad \vee\ v_1 = \textbf{nil}\big)
\end{aligned}
$$

Notice, how the order terms for the construction of b and c elements have the same parent order term which in turn is between the order term of the a and of the d children of r.

With this example, we conclude the illustration of the translation function for composition-free XQuery expressions to CIQLog. Before a brief outlook on an extension of that translation to a larger fragment of XQuery, the next section discusses the equivalence between the translation function and the above semantics for composition-free XQuery from [21].

### 8.3.5 EQUIVALENCE

To establish equivalence between the semantics of composition-free XQuery after [21] given in Table 31 and the semantic of a CIQLOG rule generated by $\mathrm{tr}_{\mathrm{XQuery}}$ for a given composition-free XQuery expression $P$, first consider that XML forests as used in the semantics are merely a special case of CIQLOG data graphs (under the mapping from axis to data graph relations given by relation in Section 8.2.2.

Thus, we have to establish that the data graph constructed by the CIQLOG rule resulting from a translation of $P$ is isomorphic to the XML forest $F$ returned by Table 31 on $P$ if restricted to the sub-trees rooted at the binding nodes $\vec{e}$.

For simplicity, we assume in the following that $P$ is wrapped in some root element root. This avoids having to consider multiple binding nodes (and thus forests instead of trees). It also addresses that above remark, that the CIQLOG semantics is always a *new* tree, whereas Table 31 allows fragments of an original tree to be returned. However, up to node identity the two forms are equivalent.

**Theorem 8.2.** *The semantics of the* CIQLOG *expression $Q$ returned by $\mathrm{tr}_{\mathrm{XQuery}}$ for a given XQuery expression $P$ is equivalent to the semantics of $P$ under Table 31 up to node identity. Furthermore, the size of $Q$ is linear in the size of $P$.*

*Proof (Sketch).* The proof is by structural induction over the shape of an XQuery expression.

For case 1 (`()`), both semantics do not change the bindings of any variables and do not add any construction. In particular, if `<root>()</root>` is the entire program both semantics are obviously equivalent (both return a tree with single, empty node root (cf. the $\mathrm{tr}_{\mathrm{Xcerpt}}$ wrapper above and case 2 in both semantics).

For case 2 and 3, element construction, add a root node with given label around the bindings returned by the evaluation of their child expressions. The iteration is on bindings in $\vec{e}$, resp. iteration variables in $\mathcal{E}$.iter which are in all cases unchanged from the call of the semantics function to nested calls except for the iteration case (case 7), where they are extended in both semantics in the same way, see below.

For case 4, observe that both semantics construct a tree isomorphic to the one rooted at a binding $e_i$ for $x_i$. However, $[\![\ ]\!]$ returns directly $(F, [e_i])$, whereas $\mathrm{tr}_{\mathrm{XQuery}}$ returns a deep-copy of the subtree rooted at $e_i$. If we disregard node identity, however, the two subtrees are equivalent.

For case 5, the same observation holds. In addition to case 4, both semantics add restrictions to the returned bindings, as expressed by the

path expression. In the case of $\mathrm{tr}_{\mathrm{XQuery}}$ this is achieved by the call to tqc. For $[\![\ ]\!]$ we also need to consider case 9.

For case 6, the sequence of two queries, both semantics delegate the translation to recursive calls on the operands and combine the result using union resp. conjunction. Note, that the involved order terms of $\mathrm{tr}_{\mathrm{XQuery}}$ are covered in $[\![\ ]\!]$ simply by concatenating the bindings returned by the second query after the end of the bindings of the first query (see definition of $\uplus$ in Section 8.3.3.

For case 7, we observe that both semantics delegate the translation, though $\mathrm{tr}_{\mathrm{XQuery}}$ uses tqc for the **in** expression. This is nevertheless equivalent, as an **in** expression in composition-free XQuery may only contain a step. This is not true of XQuery where exactly this case can not always be translated to a single Cᴵ QLog rule. The iteration variables are in both cases extended in the same way (by the single variable bound in the **in** expression).

Analog observations hold for case 8.

If one remembers that cases 10–15 of $[\![\ ]\!]$ handle expressions that only occur in conditions (but not in general query contexts), it is easy to verify that they are equivalent to the corresponding cases of tqc.

Again it is easy to see that the size of $Q$ is linear in the size of $P$ by considering that each case translates at least one construct from $Q$ (or defers entirely to another translation function) and no duplication of subresults is introduced in any case. □

## 8.4 BEYOND COMPOSITION-FREE XQUERY

In the previous sections, we have considered two important fragments of XQuery, viz. navigational XPath and composition-free XQuery, and shown ho to translate them into $\text{C}^{\text{I}}\text{QLog}^{\text{NR}}$.

We have chosen composition-free XQuery as it limits all relations to nodes of the input tree, just like in $\text{C}^{\text{I}}\text{QLog}^{\text{NR}}$. However, in full Cᴵ QLog we can also query invented nodes resulting from a previous rule application. We can exploit this to translate a larger fragment of query, viz. XQuery without user-defined functions. Intuitively, each expression is partitioned into a sequence of rules such that a following rule depends only on preceding ones. Whenever we construct data, that is then queried, we introduce a new rule.

Incidentally, this approach is used and described in more detail in [148] where a translation from XQuery to Xcerpt is defined. In fact, we can use that translation to first create an Xcerpt program for a given XQuery expression and then translate that Xcerpt program to Cᴵ QLog as described

in Section 8.

In both cases, the resulting CIQLOG program is non-recursive since each rule depends only on rules generated from nested XQuery expressions. This coincides with results from [144] where it is shown that XQuery without user-defined functions and deep-equal has NEXPTIME-complete query complexity, just as non-recursive CIQLOG (recall that non-recursive CIQLOG has the same complexity as non-recursive logic programming).

Finally, for full XQuery we have to consider user-defined, possibly recursive functions and the full operator library of XQuery [149]. The prior can be translated by the same scheme as above, but now a rule may depend also on the results of itself or rules generated from superordinate expressions. The latter can be provided as predefined relations or, where possible, implemented as CIQLOG rules, and does not add to the expressiveness of full CIQLOG.

## 8.5 CONCLUSION

CIQLOG is designed as an abstraction of the core aspects of Web query languages. Its non-recursive fragment $\text{CIQLOG}^{\text{NR}}$ is well-suited to specify and implement diverse query languages without query composition. In this chapter, we show how navigational XPath and a large fragment of XQuery, composition-free XQuery as defined in [144], can be translated to CIQLOG while preserving the standard semantics.

The translation also highlights one of the core differences between XPath (and, to some extent, Xcerpt) and XQuery: Where we can consider navigational XPath without caring about the precise iteration order and consider the semantics of XPath-expressions as sets of nodes, XQuery provides a much greater control over the iteration order on binding nodes and the order of constructed elements in the result. Though this certainly is beneficial in some cases, there are many queries for which such precise control is pointless. This has been recognized in the design of XQuery through the addition of the **unordered** operator which, essentially, switches from sequence- to (multi-) set-based semantics. Xcerpt takes the dual approach: We assume that in most cases the query author is not all that much concerned about the order of result elements. If that is not the case, a grouping expression may be adorned by an explicit **order-by** clause. Another reason for the more involved order terms in the translation of XQuery compared to Xcerpt is that we specifically disallow lists of terms in Xcerpt grouping expressions in Chapter 7, though Xcerpt 2.0 provides these expressions. If we allow such expressions as in, e.g., `r[a, all(b, c)group-by(var X), d]`, we arrive at similarly involved order terms as in the translation of XQuery

presented here.

Together with the CLQCAG algebra the above translation gives as an efficient and in many cases optimal evaluation of composition-free XQuery (and navigational XPath). For composition-free XQuery on tree, forest, or even CIG data, we obtain $\mathcal{O}(q \cdot n)$ time and space bounds if the query is tree shaped and $\mathcal{O}(n^{q_g} + q \cdot n)$ where $q$ is the size of the query, $n$ the size of the data and $q_g$ the number of answer variables or variables with non-tree edges in the query. On XML data, the space bound is even $\mathcal{O}(q \cdot d)$, cf. Section 12.9. The detailed complexities of CLQCAG are discussed in Part IV. The translation from CLQLOG to CLQCAG in Chapter 13.

# 9 TRANSLATING SPARQL



## 9.1 INTRODUCTION

Compared with XML query languages, the field of RDF query languages is less mature and has not received as much attention from research. Recently, the W3C has started to derive a standard RDF query language, called SPARQL [183], that is, visibly influenced by languages such as RDQL [163], RQL[138], and SeRQL[46], aiming to create a stable foundation for use, implementation, and research on RDF databases and query languages. Fundamentally, SPARQL is a fairly simple query language in the spirit of basic subsets of SQL or OQL. However, the specifics of RDF have lead to a number of unusual features that, arguably, make SPARQL more suited to RDF querying than previous approaches such as RDQL [163]. However, the price is a more involved semantics complemented by a tendency in [183] to redefine or ignore established notions from relational and XML query languages rather than build upon them.

Nevertheless, SPARQL is expected to become the "lingua franca" of RDF querying and thus well worth further investigation. In the following sections, we first briefly introduce into SPARQL and its semantics (based on [179] and [181] but extended to full SPARQL queries rather than only patterns). From the discussion of the semantics, we turn to the translation from SPARQL to $\text{C|QLog}^{\text{NR}}$ which turns out to be closely aligned with the semantics of [179]. The translation is also the first purely logical semantics for SPARQL which hints at how to integrate SPARQL with rule-based reasoning approaches prolific on the Semantic Web. We briefly discuss along one such approach, viz. RDFLog, the effects of extending SPARQL with rules on the translation to C|QLog.

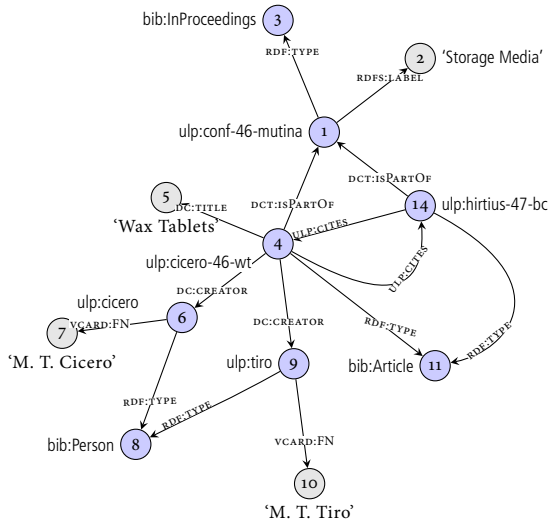



Figure 39. Exemplary Data Graph: RDF Conference Data (simplification of Figure 26 omitting the sequence container and edge positions)

## 9.2 SPARQL SYNTAX AND SEMANTICS IN 1000 WORDS

EXAMPLE. In Section 5.4, we introduce RDF and its data model together with a mapping to CIQLOG data graphs. Recall, the sample data used there, which is depicted again in Figure 39 and represents articles in a given conference and their authors.

The following SPARQL query selects from that graph all articles created by someone with the full-name "M. T. Cicero" and returns a new graph where the dc:creator relation of the original graph is inverted to my:published.[1]

```
CONSTRUCT { ?p my:published ?a }
WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
        AND ?p vcard:FN 'M. T. Cicero' }
```

The query illustrates SPARQLs fundamental query construct: a pattern $(s, p, o)$ for RDF triples (whose components are usually thought of as subject, predicate, object). Any RDF triple is also a triple pattern, but triple patterns allow variables for each component. Furthermore, SPARQL also allows literals in subject position, anticipating the same change also in RDF

1 Here, and in the following we use namespace prefixes to abbreviate IRIs. The usual IRIs are assumed for rdf, rdfs, dc (dublin core), foaf (friend-of-a-friend), vcard vocabularies. my is a prefix bound to an arbitrary IRI.

itself. We use the variant syntax for SPARQL discussed in [179] to ease the definition of syntax and semantics of the language. For instance, standard SPARQL, uses `.` instead of `AND` for triple conjunction. We consider two forms of SPARQL queries, viz. **`SELECT`** queries that return list of variable bindings and **`CONSTRUCT`** queries that return new RDF graphs. Triple patterns contained in a **`CONSTRUCT`** clause (or "template") are instantiated with the variable bindings provided by the evaluation of the triple pattern in the **`WHERE`** clause. We omit named graphs and assume that all queries are on the single input graph. An extension of the discussion to named graphs is easy (and partially demonstrated in [181]) but only distracts from the salient points of the discussion.

The full grammar of SPARQL queries as considered here (extending [179] by **`CONSTRUCT`** queries) is as follows:

⟨*query*⟩ ::= 'CONSTRUCT' ⟨*template*⟩ 'WHERE' ⟨*pattern*⟩
| 'SELECT' ⟨*variable*⟩+ 'WHERE' ⟨*pattern*⟩

⟨*template*⟩ ::= ⟨*triple*⟩ | ⟨*template*⟩ 'AND' ⟨*template*⟩ | '{' template '}'

⟨*triple*⟩ ::= ⟨*resource*⟩',' ⟨*predicate*⟩',' ⟨*resource*⟩

⟨*resource*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*blank*⟩

⟨*predicate*⟩ ::= ⟨*iri*⟩ | ⟨*variable*⟩

⟨*variable*⟩ ::= '?' ⟨*identifier*⟩

⟨*pattern*⟩ ::= ⟨*triple*⟩ | '{' ⟨*pattern*⟩ '}'
| ⟨*pattern*⟩ 'FILTER' '(' ⟨*condition*⟩ ')' |
| ⟨*pattern*⟩ 'AND' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'UNION' ⟨*pattern*⟩
| ⟨*pattern*⟩ 'MINUS' ⟨*pattern*⟩ | ⟨*pattern*⟩ 'OPT' ⟨*pattern*⟩

⟨*condition*⟩ ::= ⟨*variable*⟩ '=' ⟨*variable*⟩ | ⟨*variable*⟩ '=' (⟨*literal*⟩|⟨*iri*⟩)
| 'BOUND(' ⟨*variable*⟩ ')' | 'isBLANK(' ⟨*variable*⟩ ')'
| 'isLITERAL(' ⟨*variable*⟩ ')' | 'isIRI(' ⟨*variable*⟩ ')'
| ⟨*negation*⟩ | ⟨*conjunction*⟩ | ⟨*disjunction*⟩

⟨*negation*⟩ ::= '¬'⟨*condition*⟩

⟨*conjunction*⟩ ::= ⟨*condition*⟩ '∧' ⟨*condition*⟩

⟨*disjunction*⟩ ::= ⟨*condition*⟩ '∨' ⟨*condition*⟩

We pose some additional syntactic restrictions: SPARQL queries are *range-restricted*, i.e., all variables in the "head" (**`CONSTRUCT`** or **`SELECT`** clause) also occurs in the "body" (**`WHERE`** clause) of the query. We assume *error-free* SPARQL expressions (in contrast to [179] and [181]), i.e., for each **`FILTER`** expression all variables occurring in the (right-hand) condition must also occur in the (left-hand) pattern. The first limitation is as in standard SPARQL, the second is allowed in standard SPARQL but can easily

recognized a-priori and rewritten to the canonical false **FILTER** expression (as **FILTER** expressions with unbound variables raise errors which, in turn, are treated as a false filter, see "effective boolean value" in [183].

Finally, we allow only *valid RDF constructions* in **CONSTRUCT** clauses, i.e., no literal may occur as a subject, all variables occurring in subject position are never bound to literals, and all variables occurring in predicate position are only ever bound to IRIs (but not to literals or blank nodes). The first condition can be enforced statically, the others by adding appropriate `isIRI` or negated isLITERAL filters to the query body.

Following [181], we define the semantics of SPARQL queries based on *substitutions*. A substitution $\theta = \langle v_1, n_1, \ldots, v_k : n_k \rangle$ with $v_i \in \mathsf{Vars}(Q) \wedge n_i \in nodes(D)\}$ for a query $Q$ over a data graph $D$ (as in Section 5.4) maps some variables from $Q$ to nodes in $D$. For a substitution $\theta$ we denote with $\mathrm{dom}(\theta)$ the variables mapped by $\theta$. Given a triple pattern $t = (s, p, o)$, we denote with $t\theta$ the application of $\theta$ to $t$ replacing all occurrences of variables mapped in $\theta$ by their mapping in $t$. For a triple $(s, p, o)$ containing no variables, we say $(s, p, o) \in D$ if there is a $p$ labeled edge between $s$ and $o$ labeled nodes in $D$.

On sets of substitutions the usual relational operations $\bowtie$, $\cup$, and $\setminus$ apply. We define the (left) semi-join $R \ltimes S = (R \bowtie S) \cup (R \setminus S)$.

Finally, given a template $t$, i.e., a conjunction of triple patterns, $\mathsf{std}(t)$ returns $t$ but replacing each blank node identifier (i.e., strings of the form `_:`*identifier*) with a new blank node identifier not occurring in $D$ and not created by a prior application of std. Intuitively, $\mathsf{std}(t)$ creates a new instance of $t$ such that the blank nodes of two instances (and any instance with the input graph) do not overlap.

Using these definitions, Table 37 gives the semantics of SPARQL **SELECT** and **CONSTRUCT** queries by means of $[\![\ ]\!]^D$. $[\![\ ]\!]^D$ translates the **WHERE** clause using $[\![\ ]\!]^D_{\mathsf{Subst}}$ and a **CONSTRUCT** clause, if present, using $[\![\ ]\!]^D_{\mathsf{Graph}}$. For a **SELECT** query, we project the set of substitutions returned by $[\![\ ]\!]^D_{\mathsf{Subst}}$ to the set of answer variables $V$. For a **CONSTRUCT** query we apply each substitution $\theta \in [\![ P ]\!]^D_{\mathsf{Subst}}$ to a new instance of the template $t$ contained in the **CONSTRUCT** clause created using std. Applying a substitutions is straightforward except that triples containing one or more variables that bound to **nil** by $\theta$ are omitted entirely.

The semantics of a SPARQL pattern $P$ contained in the **WHERE** clause is given by $[\![ P ]\!]^D$ and produces a set of substitutions (or bindings) for variables in $P$. Triple patterns $t$ (case 1) are evaluated to the set of substitutions $\theta$ such that the $t\theta$ contains no more variables and falls in $D$. Pattern compositions **AND**, **UNION**, **MINUS**, and **OPT** are reduced to the appropriate operations on sets of substitutions (cases 2–4). **FILTER** expressions (case 5) are again evaluated straightforwardly, as restrictions on the substitutions returned

$$
\begin{array}{ll}
[\![ (s,p,o) ]\!]^D_{\mathsf{Subst}} & = \{\theta : \mathrm{dom}(\theta) = \mathsf{Vars}((s,p,o)) \wedge t\theta \in D\} \\
\left[\!\!\left[ \mathit{pattern}_1 \ \texttt{AND}\ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} & = \left[\!\!\left[ \mathit{pattern}_1 \right]\!\!\right]^D_{\mathsf{Subst}} \bowtie \left[\!\!\left[ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} \\
\left[\!\!\left[ \mathit{pattern}_1 \ \texttt{UNION}\ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} & = \left[\!\!\left[ \mathit{pattern}_1 \right]\!\!\right]^D_{\mathsf{Subst}} \cup \left[\!\!\left[ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} \\
\left[\!\!\left[ \mathit{pattern}_1 \ \texttt{MINUS}\ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} & = \left[\!\!\left[ \mathit{pattern}_1 \right]\!\!\right]^D_{\mathsf{Subst}} \setminus \left[\!\!\left[ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} \\
\left[\!\!\left[ \mathit{pattern}_1 \ \texttt{OPT}\ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} & = \left[\!\!\left[ \mathit{pattern}_1 \right]\!\!\right]^D_{\mathsf{Subst}} \ltimes \left[\!\!\left[ \mathit{pattern}_2 \right]\!\!\right]^D_{\mathsf{Subst}} \\
[\![ \mathit{pattern}\ \texttt{FILTER}\ \mathit{condition} ]\!]^D_{\mathsf{Subst}} & = \{\theta \in [\![ \mathit{pattern} ]\!]^D_{\mathsf{Subst}} : \mathsf{Vars}(\mathit{condition}) \subset \mathrm{dom}(\theta) \\
 & \qquad \wedge\, [\![ \mathit{condition} ]\!]^D_{\mathsf{Bool}}(\theta)\}
\end{array}
$$

---

$$
\begin{array}{ll}
[\![ \mathit{condition}_1 \wedge \mathit{condition}_2 ]\!]^D_{\mathsf{Bool}}(\theta) & = [\![ \mathit{condition}_1 ]\!]^D_{\mathsf{Bool}}(\theta) \wedge [\![ \mathit{condition}_2 ]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![ \mathit{condition}_1 \vee \mathit{condition}_2 ]\!]^D_{\mathsf{Bool}}(\theta) & = [\![ \mathit{condition}_1 ]\!]^D_{\mathsf{Bool}}(\theta) \vee [\![ \mathit{condition}_2 ]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![ \neg \mathit{condition} ]\!]^D_{\mathsf{Bool}}(\theta) & = \neg [\![ \mathit{condition} ]\!]^D_{\mathsf{Bool}}(\theta) \\
[\![ \texttt{BOUND}(?v) ]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \neq \mathbf{nil} \\
[\![ \texttt{isLITERAL}(?v) ]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in L \\
[\![ \texttt{isIRI}(?v) ]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in I \\
[\![ \texttt{isBLANK}(?v) ]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta \in B \\
[\![ ?v = \mathit{literal} ]\!]^D_{\mathsf{Bool}}(\theta) & = v\theta = \mathit{literal} \\
[\![ ?u = ?v ]\!]^D_{\mathsf{Bool}}(\theta) & = u\theta = v\theta \wedge u\theta \neq \mathbf{nil}
\end{array}
$$

---

$$
\begin{array}{ll}
[\![ \mathit{triple} ]\!]^D_{\mathsf{Graph}}(\theta) & = \mathit{triple}\theta \text{ if } \forall v \in \mathsf{Vars}(\mathit{triple}) : v\theta \neq \mathbf{nil}, \top \text{ otherwise} \\
\left[\!\!\left[ \mathit{template}_1 \ \texttt{AND}\ \mathit{template}_2 \right]\!\!\right]^D_{\mathsf{Graph}}(\theta) & = \left[\!\!\left[ \mathit{template}_1 \right]\!\!\right]^D_{\mathsf{Graph}}(\theta) \cup \left[\!\!\left[ \mathit{template}_2 \right]\!\!\right]^D_{\mathsf{Graph}}(\theta) \\
[\![ \texttt{CONSTRUCT}\ t\ \texttt{WHERE}\ p ]\!]^D & = \bigcup_{\theta \in [\![ P ]\!]^D_{\mathsf{Subst}}} [\![ \mathsf{std}(t) ]\!]^D_{\mathsf{Graph}}(\theta) \\
[\![ \texttt{SELECT}\ V\ \texttt{WHERE}\ p ]\!]^D & = \pi_V([\![ P ]\!]^D_{\mathsf{Subst}})
\end{array}
$$

Table 37. Semantics for SPARQL

by the (left-hand) pattern with the boolean formula that is provided by $[\![\ ]\!]^D_{\mathsf{Bool}}$ for the condition of the filter expression. $\mathsf{Vars}(\mathit{condition}) \subset \mathrm{dom}(\theta)$ is not strictly necessary as it merely restates that we only consider error-free SPARQL queries.

## 9.3 TRANSLATING SPARQL QUERIES

Translating SPARQL to CIQLog is, for the most part, a direct mirror of the semantics in Table 37. The main difference is when translating **CONSTRUCT** clauses. Here, we create value invention terms for each resource in the template that depend on all variables in the **CONSTRUCT** clause. This implements std in the above semantics, i.e., the instantiation of the blank nodes for each substitution (i.e., binding tuple). This implies, however, that the result is not the data graph representation of an RDF graph in the sense of Section 5.4 since it may contain several nodes with the same (IRI or literal) label. he translation of **WHERE** clauses, though, does not use repeated variable occurrences in the body but separate variables for each subject, predicate, or object and label variables for common occurrences of the same SPARQL variable. Thus it can not distinguish between a graph with several nodes with the same label (each carrying some but not all properties of the named resource) and one where all these nodes are collapsed. Nevertheless, we may want to create a proper data graph representation by collapsing all such nodes. This can be achieved in CIQLog by a simple graph transformation on all nodes with the same label that exploits that value invention in CIQLog can be parametrized with an equivalence relation. Usually, we assume node equality $\doteq$, but in this case we employ label equality $\cong$ and thus create only one node in the transformed graph per unique node or edge label in the input graph.

TRANSLATION EXAMPLES. To illustrate this point and the general translation more closely, consider again the above SPARQL example:

```
CONSTRUCT { ?p my:published ?a }
WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
        AND ?p vcard:FN 'M. T. Cicero' }
```

In CIQLog, we can express the same query constructing a new graph with two nodes and one edge per binding tuple for p and a. Here and in the following examples we omit conditionals if all variables are non-optional and use prefix abbreviations also for CIQLog labels:

$$\mathfrak{L}(\mathsf{id}_1(\mathcal{A}), v_p) \wedge \circ\!\!\rightarrow(\mathsf{id}_1(\mathcal{A}), \mathsf{id}_2(\mathcal{A})) \wedge \mathfrak{L}(\mathsf{id}_2(\mathcal{A}), \texttt{my:published}) \wedge \rightarrow\!\!\circ(\mathsf{id}_3(\mathcal{A}), \mathsf{id}_2(\mathcal{A})) \wedge \mathfrak{L}(\mathsf{id}_3(\mathcal{A}), v_a) \longleftarrow \mathfrak{L}(s_1, v_a) \wedge \circ\!\!\rightarrow(s_1, p_1) \wedge \mathfrak{L}(p_1, \texttt{rdf:type}) \wedge$$

```
        →○(o1, p1) ∧ 𝔏(o1,bib:Article) ∧
5     𝔏(s2,va) ∧ ○→(s2,p2) ∧ 𝔏(p2,dc:creator) ∧ →○(o2, p2) ∧ 𝔏
          (o2,vp) ∧
      𝔏(s3,vp) ∧ ○→(s3,p3) ∧ 𝔏(p3,vcard:FN) ∧ →○(o3,p3) ∧
7       𝔏(o3,'M. T. Cicero')
```

The above CIQLOG rule illustrates the general translation scheme: each triple pattern in the body is separately translated using new variables for subject, predicate, and object. If any of those is a SPARQL variable, we add a label restriction on the respective variable and a label variable that is the associated CIQLOG variable of that SPARQL variable. In the head, triple patterns are also translated independently, but now label restrictions are established with query variables (line 2). All node value expressions in the head are over all variables in the CONSTRUCT clause (abbreviated above as $\mathcal{A} = [v_a, v_p]$). This includes possible optional variables (see next example). As equivalence relation we use here equality on labels, not node identity.

In contrast to Xcerpt, XPath, and XQuery, we do not include root relations though if desired appropriate relations (e.g., to each named resource) can be easily added.

The second example focuses on the effect of **OPT** clauses and blank nodes in the head:

```
1 CONSTRUCT { _:group my:member ?a AND ?a my:otherAuthor ?p2 }
  WHERE { ?a rdf:type bib:Article AND ?a dc:creator ?p
3         AND ?p vcard:FN 'M. T. Cicero'
          OPT { ?a dc:creator ?p2 FILTER (¬ ?p = ?p2) } }
```

We select in addition any further creators of a paper authored by Cicero. We return a graph over the article and the optional further creator. `_:group` is a blank node (with local identifier `group`. In CIQLOG we obtain the following rule for this SPARQL query (with $\mathcal{A} = [v_a, v_p, v_{p_2}]$)

```
𝔏(id1(𝒜), _:id1(𝒜)) ∧ ○→(id1(𝒜),id2(𝒜)) ∧ 𝔏(id2(𝒜),my:member) ∧
2     →○(id3(𝒜),id2(𝒜)) ∧ 𝔏(id3(𝒜),va) ∧
      if vp ≠ nil ∧ p2 ≠ nil then
4       𝔏(id4(𝒜),va) ∧ ○→(id4(𝒜),id5(𝒜)) ∧ 𝔏
            (id5(𝒜),my:otherAuthor) ∧
        →○(id6(𝒜),id5(𝒜)) ∧ 𝔏(id6(𝒜),vp2) ←
6   𝔏(s1,va) ∧ ○→(s1,p1) ∧ 𝔏(p1,rdf:type) ∧ →○(o1, p1) ∧ 𝔏
      (o1,bib:Article) ∧
    𝔏(s2,va) ∧ ○→(s2,p2) ∧ 𝔏(p2,dc:creator) ∧ →○(o2,p2) ∧ 𝔏(o2,vp) ∧
8   𝔏(s3,vp) ∧ ○→(s3,p3) ∧ 𝔏(p3,vcard:FN) ∧ →○(o3,p3) ∧ 𝔏
      (o3,'M.T.Cicero') ∧
  (𝔏(s4,va) ∧ ○→(s4,p4) ∧ 𝔏(p4,dc:creator) ∧ →○(o4,p4) ∧
10 𝔏(o4,vp2) ∧ ¬(vp ≠ vp2) ∨ s4 = nil ∧
   p4 = nil ∧ o4 = nil ∧ vp2 = nil)
```

First, in the body the optional is realized by an disjunction where the second part sets all newly introduced variables of the sub-expression to **nil**, thus ensuring that the query never fails due to the optional part (but the variables are **nil** in case of failure). In the head, we add a conditional around all triple pattern containing optional variables. Notice, how we use value invention to instantiate a new blank node for each binding pair in line 1.

| function | SPARQL expression | CIQLog expression |
|---|---|---|
| $\text{tr}_{\text{SPARQL}}\langle$CONSTRUCT *template* WHERE *pattern*$\rangle$ | | $= C \longleftarrow Q$ **where** $(C, v) = \text{tsh}(\mathcal{E}')\langle \textit{template} \rangle$<br>$(\mathcal{E}, Q) = \text{tsp}(\varnothing)\langle \textit{pattern} \rangle$<br>$\mathcal{E}' = \mathcal{E}$ with $\mathcal{E}'$.iter = Vars(*template*) |
| $\text{tr}_{\text{SPARQL}}\langle$SELECT $v_1, \ldots, v_n$ WHERE *pattern*$\rangle$ | | $= \text{ans}(\mathcal{E}(v_1), \ldots, \mathcal{E}(v_n)) \longleftarrow Q$<br>**where** $(\mathcal{E}, Q) = \text{tsp}(\varnothing)\langle \textit{pattern} \rangle$ |
| $\text{tsh}(\mathcal{E})\langle \textit{literal} \rangle$ | | $= (\mathfrak{L}(v, \text{'}\textit{literal}\text{'}), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier |
| $\text{tsh}(\mathcal{E})\langle \textit{iri} \rangle$ | | $= (\mathfrak{L}(v, \textit{iri}), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier |
| $\text{tsh}(\mathcal{E})\langle ?\textit{vid} \rangle$ | | $= (\mathfrak{L}(v, \mathcal{E}(\textit{vid})), v)$ **where** $v = \text{id}(\mathcal{E}.\text{iter})$ with id new identifier |
| $\text{tsh}(\mathcal{E})\langle (s, p, o) \rangle$ | | $= (\,$ **if** $C$ **then** $F_s \wedge F_p \wedge F_o \wedge \circ\!\!\rightarrow(v_s, v_p) \wedge \rightarrow\!\!\circ(v_o, v_p), v_s)$<br>**where** $C = (v_1 \neq \textbf{nil} \wedge \ldots \wedge v_k \neq \textbf{nil})$ where $\{v_1, \ldots, v_k\} = \text{Vars}((s, p, o))$<br>$(F_s, v_s) = \text{tsh}(\mathcal{E})\langle s \rangle \qquad (F_p, v_p) = \text{tsh}(\mathcal{E})\langle p \rangle$<br>$(F_o, v_o) = \text{tsh}(\mathcal{E})\langle o \rangle$ |
| $\text{tsh}(\mathcal{E})\langle t_1$ AND $t_2 \rangle$ | | $= (F_1 \wedge F_2, v_2)$ **where** $(F_1, v_1) = \text{tsh}(\mathcal{E})\langle t_1 \rangle$<br>$(F_2, v_2) = \text{tsh}(\mathcal{E})\langle t_2 \rangle$ |

Table 39. Translating SPARQL queries and **CONSTRUCT** clauses

TRANSLATION FUNCTION. Guided by these examples, we can turn to the translation function for SPARQL, $\text{tr}_{\text{SPARQL}}$, as specified in Table 39. It employs two helper functions, tsh for translating **CONSTRUCT** clauses and tsp for translating **WHERE** clauses. As for the translation of Xcerpt and XQuery we use an environment containing mapping from SPARQL variables to those of CIQLog and a list of iteration variables that is always the set of all variables occurring in the **CONSTRUCT** clause (case 1 in Table 39). Notice the use of NewVars in the translation of **OPT** that retrieves all variables introduced in the sub-pattern under a **OPT** (i.e., variables not retrieved from the environment). These variables are then used in the second part of the disjunction.

| function SPARQL expression | CIQLog expression |
|---|---|
| $\mathrm{tsr}(\mathcal{E})\langle \mathit{literal}\rangle$ | $= (\mathcal{E}, \mathfrak{L}(v, \text{'}\mathit{literal}\text{'}), v)$ **where** $v$ is a new variable |
| $\mathrm{tsr}(\mathcal{E})\langle \mathit{iri}\rangle$ | $= (\mathcal{E}, \mathfrak{L}(v, \mathit{iri}), v)$ **where** $v$ is a new variable |
| $\mathrm{tsr}(\mathcal{E})\langle ?\mathit{vid}\rangle$ | $= (\mathcal{E} +\!\!+ \{(\mathit{vid}, l)\}, \mathfrak{L}(v, l), v)$ **where** $v$ is a new variable and $l = \mathcal{E}(\mathit{vid})$ if defined, otherwise a new variable |
| $\mathrm{tsp}(\mathcal{E})\langle (s, p, o)\rangle$ | $= (\mathcal{E}_o, F_s \wedge F_p \wedge F_o \wedge \circ\!\!\rightarrow(v_s, v_p) \wedge \rightarrow\!\!\circ(v_o, v_p))$ **where** $(\mathcal{E}_s, F_s, v_s) = \mathrm{tsr}(\mathcal{E})\langle s\rangle$ $(\mathcal{E}_p, F_p, v_p) = \mathrm{tsr}(\mathcal{E}_s)\langle p\rangle$ $(\mathcal{E}_o, F_o, v_o) = \mathrm{tsr}(\mathcal{E}_p)\langle o\rangle$ |
| $\mathrm{tsp}(\mathcal{E})\langle p_1 \texttt{ AND } p_2\rangle$ | $= (\mathcal{E}_2, F_1 \wedge F_2)$ **where** $(\mathcal{E}_1, F_1) = \mathrm{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathrm{tsp}(\mathcal{E}_1)\langle p_2\rangle$ |
| $\mathrm{tsp}(\mathcal{E})\langle p_1 \texttt{ UNION } p_2\rangle$ | $= (\mathcal{E}_2, F_1 \vee F_2)$ **where** $(\mathcal{E}_1, F_1) = \mathrm{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathrm{tsp}(\mathcal{E}_1)\langle p_2\rangle$ |
| $\mathrm{tsp}(\mathcal{E})\langle p_1 \texttt{ MINUS } p_2\rangle$ | $= (\mathcal{E}_2, F_1 \wedge \neg(F_2))$ **where** $(\mathcal{E}_1, F_1) = \mathrm{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathrm{tsp}(\mathcal{E}_1)\langle p_2\rangle$ |
| $\mathrm{tsp}(\mathcal{E})\langle p_1 \texttt{ OPT } p_2\rangle$ | $= (\mathcal{E}_2, F_1 \wedge (F_2 \vee F_3)$ **where** $(\mathcal{E}_1, F_1) = \mathrm{tsp}(\mathcal{E})\langle p_1\rangle$ $(\mathcal{E}_2, F_2) = \mathrm{tsp}(\mathcal{E}_1)\langle p_2\rangle$ $F_3 = \bigvee_{v \in \mathrm{NewVars}(F_1) \cup \mathrm{NewVars}(F_2)} (v = \texttt{nil})$ |
| $\mathrm{tsp}(\mathcal{E})\langle p_1 \texttt{ FILTER } p_2\rangle$ | $= (\mathcal{E}_1, F_1 \wedge \mathrm{tsc}(\mathcal{E}_1)\langle p_2\rangle)$ **where** $(\mathcal{E}_1, F_1) = \mathrm{tsp}(\mathcal{E})\langle p_1\rangle$ |
| $\mathrm{tsc}(\mathcal{E})\langle c_1 \wedge c_2\rangle$ | $= \mathrm{tsc}(\mathcal{E})\langle c_1\rangle \wedge \mathrm{tsc}(\mathcal{E})\langle c_2\rangle$ |
| $\mathrm{tsc}(\mathcal{E})\langle c_1 \vee c_2\rangle$ | $= \mathrm{tsc}(\mathcal{E})\langle c_1\rangle \vee \mathrm{tsc}(\mathcal{E})\langle c_2\rangle$ |
| $\mathrm{tsc}(\mathcal{E})\langle \neg c\rangle$ | $= \neg(\mathrm{tsc}(\mathcal{E})\langle c\rangle)$ |
| $\mathrm{tsc}(\mathcal{E})\langle ?\mathit{vid}_1 = ?\mathit{vid}_2\rangle$ | $= \mathcal{E}(\mathit{vid}_1) = \mathcal{E}(\mathit{vid}_2)$ |
| $\mathrm{tsc}(\mathcal{E})\langle ?\mathit{vid} = \mathit{literal}\rangle$ | $= \mathcal{E}(\mathit{vid}_1) = \text{'}\mathit{literal}\text{'}$ |
| $\mathrm{tsc}(\mathcal{E})\langle ?\mathit{vid} = \mathit{iri}\rangle$ | $= \mathcal{E}(\mathit{vid}) = \mathit{iri}$ |
| $\mathrm{tsc}(\mathcal{E})\langle \texttt{BOUND}(?\mathit{vid})\rangle$ | $= \mathcal{E}(\mathit{vid}) \neq \texttt{nil}$ |
| $\mathrm{tsc}(\mathcal{E})\langle \texttt{isBLANK}(?\mathit{vid})\rangle$ | $= \mathcal{E}(\mathit{vid}) = \texttt{\_:}\langle \mathit{identifier}\rangle$ |
| $\mathrm{tsc}(\mathcal{E})\langle \texttt{isLITERAL}(?\mathit{vid})\rangle$ | $= \text{'}\langle \mathit{string}\rangle\text{'}$ |
| $\mathrm{tsc}(\mathcal{E})\langle \texttt{isIRI}(?\mathit{vid})\rangle$ | $= \neg(\mathrm{tsc}(\mathcal{E})\langle \texttt{isLITERAL}(?\mathit{vid})\rangle \vee \mathrm{tsc}(\mathcal{E})\langle \texttt{isBLANK}(?\mathit{vid})\rangle)$ |

Table 41. Translating SPARQL patterns and conditions

The translation of SPARQL construct patterns is fairly unremarkable. Notice that variables are translated by retrieving the related label variable from the environment.

SPARQL patterns are translated using tsp, specified in Table 41, and its helpers tsr (for resources) and tsc (for conditions). Subjects, predicates, and objects are translated by always creating new variables and placing label restrictions on those variables. When translating SPARQL variables, we establish label restrictions with the associated label variable of CIQLog ($l$ in case 3 of Table 41). Otherwise the translation is fairly straightforward.

We conclude the translation of SPARQL with the conjecture that the graph constructed by $\mathsf{tr}_{\mathsf{SPARQL}} P$ contains (in the sense defined above), up to consistent renaming of blank nodes, the same triples as $[\![ P ]\!]^D$.

*Conjecture* 9.1. For a given SPARQL **CONSTRUCT** query $P$ there is a mapping $f$ on literals, IRIs, and blank nodes, that is the identity on literals and IRIs, such a triple $(s, p, o) \in T$ if and only if $(f(s), f(p), f(o)) \in [\![ P ]\!]^D$ where $T$ is in the graph obtained from the evaluation of $\mathsf{tr}_{\mathsf{SPARQL}} P$.

## 9.4 FROM SPARQL TO RULES: RDFLOG

SPARQL queries can be considered as non-recursive, single-rule expressions. There have been some approaches to extending SPARQL with rules, e.g., [181], or to embed SPARQL queries in a rule-based query language.

The above translation yields a single CIQLog rule for each SPARQL query. Obviously, we can allow a program to contain many such rules which then can provide input to each other.

However, there are a number of challenges when adding rules to an RDF query language that are better addressed in RDFLog [63]. Most notably, SPARQL's heads always group over all answer variables which limits the kind of queries that can be expressed significantly (at no gain in complexity as shown in [63].

With the same observation as for the previous languages, it is easy to see from the translation functions that the following result holds

**Theorem 9.1.** *The size of the* CIQLog *expression $Q$ returned by $\mathsf{tr}_{\mathsf{SPARQL}}$ for a given SPARQL query $P$ is linear in the size of $P$.*

## 9.5 CONCLUSION

Compared with the translations for XQuery and Xcerpt, SPARQL is an easy target for translation to CIQLog. The main difficulty lies in the graph construction not in the query patterns. We have employed a translation

scheme using label variables above. This emphasizes that SPARQL's domain is better thought of as IRIs, literals, and blank node identifiers. This deviates from the common view of the domain of an RDF graph as nodes, as the suggested in the RDF model theory. However, it coincides with the perspective on RDF in [179] and [181]. For the most part, we could nevertheless also employ a node-based translation scheme but then the result construction becomes far more involved.

Part IV

# THEORY. CIQCAG: SCALING FROM TREES TO GRAPHS

# 10 PRINCIPLES AND MOTIVATION



## 10.1 INTRODUCTION

*Structure is a central feature of data*

We assign meaning to things by enumerating their features (or properties or attributes) and placing them in relation with other things. This enables us to distinguish, classify, and, eventually, act upon such things. The same applies to digital data items: to find, analyse, classify, and, eventually, use as basis for actions we need to place data items in relation to other data items: A book to its author, a bank transaction to the bank, the source and the target of the transaction, a patient to its treatment history, its doctor, etc.

How we describe these relations between data items (as well as their features) is the purview of data models. Recently the relational data model, tailored to relations of arbitrary shape, has been complemented by semi-structured data models tailored to Web data. What sets these data models apart from relational data is an even greater focus on relations or links while delegating features or attributes to second-class citizens or dropping them entirely, as in RDF. At the same time most of these data models share a strong hierarchical bias: XML is most often considered tree data.[1] RDF

1 Though ID-links justify a more graph-like view of XML.

and other ontology languages allow arbitrary graphs but ontologies often have dominantly hierarchical "backbones", formed, e.g., by subclass or part-of relations. To summarize, *structure* is a central property of data and data models determine what shape those structures may take.

*Structure is also a central feature of queries*

When we want to actually *do* something with the data represented in any of these data models, we use queries. Again, exploiting the relations among sought-for data items is essential: to find all authors of books on a given topic, to find the bank with the highest transaction count, to identify an illness by analysing patterns in a patient's health records. Thus queries mirror the structure of the sought-for data, though often with a richer vocabulary, allowing, e.g., for recursive relation traversal or don't-care parts: a patient is chronically ill if there is some illness (we don't care which illness) that recurs regularly in that patient's health records. Queries may contain additional derived (i.e.., not explicitly represented or "extensional") relations—such as equalities or order between the value of data items. The shape of a query is, thus, not limited to the shape of the data but may contain additional relations. Only if we limit a query to extensional relations must the shape of the query mirror (a substructure) of the shape of the data. For instance, when querying tree data queries may take the only the shape of trees, if we allow only extensional relations[2], but may have arbitrary shape if we allow derived relations. To summarize, as for data, structure is a central feature of queries and mirrors the structure of the sought-for data. However, the structure of a query is linked to the structure of the query only if we consider exclusively extensional relations in the query.

*Limits are good*

The reason we should care about the shape of the data or queriesis a growing canon of approaches that obtain better complexity and performance for query evaluation if certain limits are imposed on the shape of data, queries, or both.

If we consider arbitrary data, we have little reliable means for compacting relation information. On ordered tree data, in contrast, we can use any number of encodings, e.g., interval encodings [86, 85], hierarchical or path-based labeling [173], or schemes based on structural summaries [200]. In essence, these encodings exploit that structural relations in trees follow certain rules, e.g., each node has a unique parent, the descendants of each node are among the descendants of all its ancestors, each node

2 Even allowing transitive closure on extensional relations allows already for queries where there are, e.g., two paths between two query nodes. However, as shown in [170] for XPath and in [167] for the general case of tree queries, such graph shaped queries can always be reduced to sets of tree-shaped queries—though potentially for the cost of an exponential increase in query size.

has a unique following and preceeding sibling, etc. Interval encodings on trees, e.g., allow us to compact closure relations quadratic in the tree size into a linear size interval encoding.

For queries, we can make a similar observation: if we allow arbitrary "links" in a query, we need to manage relations between bindings for all nodes in the query at once. However, the relations between the nodes may be limited, e.g., if the query is tree shaped, bindings for each node are directly related only to bindings of its "parent". In fact, if we consider the answers to a query as a relation with the nodes as columns, answers of a tree-shaped query always exhibit multivalued dependencies [91]: In fact, we can normalize or decompose such a relation for a query with $n$ nodes into $n-1$ separate relations that together faithfully represent the original relation (*lossless-join* decomposition to binary relations over each pair of adjacent variables in the query). This allows us to compact an otherwise potentially exponential answer (in the data size) into a polynomial representation.

Neither observation is particularly new: acyclic or tree queries on relational data as interesting polynomial-time subclass have been studied, e.g., in [203] and [110]. More recently, the increasing popularity of Web data such as XML triggered renewed interest and reinvestigation of tree data and tree queries as interesting restrictions of general relational structures and queries. Several novel techniques tailored to XML data and XPath or similar tree queries have shown the benefit of exploiting the hierarchical nature of the data for efficient query evaluation: polynomial twig joins [48]; XPath evaluation [113]; polynomial evaluation of tree queries against XML streams [9, 171, 168], linear tree labeling schemes [119, 200] allowing constant time access to structural closure relations such as descendant or following; and path indices [71] enabling constant time evaluation of path queries.

*Beyond trees ...*

As stated, these techniques have received considerable attention when data and queries are tree shaped. However, data often contains some non-tree aspects, even if the tree aspects are dominant, e.g., ID-links in XML or many ontologies (such as the GeneOntology [87]). Practical queries (such as XQuery or Xcerpt) often go beyond tree queries, e.g., to express value or identity joins, even if they contain a majority of structural conditions. Driven by such considerations, interest in adapting above techniques beyond trees is growing (e.g., labeling and reachability for graph data [199, 195] or hypertree decomposition for polynomial queries beyond trees [108]).

Therefore, we explore in this work means of building on the above mentioned techniques but pushing them beyond trees. We orient this exploration along the following two questions:

**(1)** Can we find an interesting and practically relevant class of (data)

graphs that is a proper superset of trees, yet to which algorithms such as twig joins [48], so far limited to trees or DAGs (directed acyclic graphs) [70], can be extended without affecting (time and space) complexity?

**(2)** Second, we observe that data and, particularly, queries are often mostly trees with only limited non-tree parts. However, any non-tree part makes most of the techniques discussed above for tree queries inapplicable. Can we integrate the above technologies in the processing of general graph queries in such a way that (often significant) hierarchical components can be evaluated using polynomial algorithms, limiting the degradation of query complexity to non-tree parts of the query?

*Our solution: CIQCAG algebra build around a novel characterization of compactable graphs and a novel data structure for representing intermediate query results*

In the following, we answer both questions essentially positively by introducing a novel algebra, called CIQCAG, the **c**ompositional, **i**nterval-based **q**uery and **c**omposition **a**lgebra for **g**raphs. CIQCAG is a fully algebraic approach to querying Web data (be it in XML, RDF, or other semi-structured shape) that is build around two central contributions:

**(1)** a novel *characterization of (data) graphs* admissible to interval-based compaction. For this new class of data, called **continuous-image graphs** (or CIGs for short), we can provide linear-space and almost linear-time algorithms for evaluating tree queries rivaling the best known algorithms for tree data.

**(2)** An algebra for a two-phase evaluation of queries separating a tree core of the query from the remaining non-tree constraints. The operations of the algebra closely mirror relational algebra (and can, in fact, be implemented in standard SQL), but **(a)** a novel *data structure* influenced by Xcerpt's memoization matrix [187, 52] and the complete answer aggregates approach [161] allows for exponentially more succinct storage of intermediate results for the tree core of a query. Together with a set of operators on this data structure, called **sequence map**, this enables CIQCAG to **(b)** evaluate almost-tree queries with nearly polynomial time limiting the degradation in performance to non-tree parts of the query. **(c)** Finally, the algebra is tailored to be agnostic of the actual realization of the used relations. This makes it particularly easy to integrate approaches for *arbitrary derived relations* and *indices* in addition to extensional relations, reachability indices such as interval labeling [116] for tree data or [195] for graph data and path indices such as DataGuides [107] or [71] for tree data.

*Expressiveness*

In the following chapters, we focus on the basic query and construction algebra which covers non-recursive, single-rule Xcerpt as well as non-compositional XQuery (as defined in [144]), see Part III. However,

in Section 13.4 we briefly discuss an extension of this algebra with an iteration operator. This extension allows the evaluation of full Xcerpt and a considerably larger fragment of XQuery than otherwise covered.

*Overview*

Before we begin the formal introduction of algebra in Chapters 11 and 12, the remainder of this chapter serves to give a first intuition of the CIQCAG algebra. First, we briefly (Section 10.2) introduce continuous-image graphs as a class of graph data where the proposed algebra performs as well as the best approaches for tree data. This is achieved using the sequence map data structure, a novel representation of intermediary results of tree queries (or tree cores of graph queries), introduced in Section 10.3. This data structure is embedded into the CIQCAG algebra in Section 10.4 where we give a first glance at the structure of the algebra and its evaluation phases. We wrap up this introduction with a brief overview of complexity results for the discussed algebra, as well as a first comparison with existing approaches in Section 10.5.

## 10.2 DATA BEYOND TREES: CONTINUOUS-IMAGE GRAPHS

Tree data, as argued above, allows us to represent relations on that data more compactly, e.g., using various interval-based labeling schemes. Here, we introduce a new class of graphs, called *continuous-image graphs* (or CIGs for short), that generalize features of tree data in such a way that we can evaluate (tree) queries on CIGs with the same time and space complexity as techniques such as twig joins [48] which are limited to tree data only.

*Characterization of continuous image graph*

Continuous-image graphs are a proper superset of (ordered) trees. On trees we require that each node has at most one parent. For continuous-image graphs, however, we only ask that we can find a single order on all nodes of the graph such that the children of each parent form a continuous interval in that order. Formally, we define a continuous-image graph by means of the image interval property (a generalization of corresponding properties of tree-shaped relations or closure relations of tree-shaped base

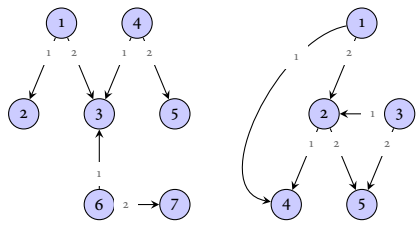



Figure 40. Sharing: On the Limits of Continuous-image Graphs

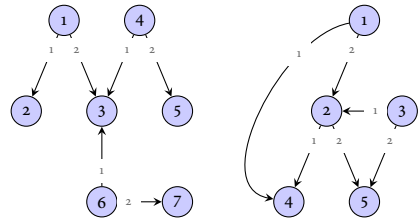



Figure 41. Sharing: On the Limits of Continuous-image Graphs

relations, as discussed in Section 11.3). Recall that we denote with $R(n)$ for a node $n \in N$ and a binary relation $R$ over the domain $N$ the set $\{n' \in N : (n, n') \in R\}$.

**Definition 10.1** (Continuous-image Graph)**.** Let $R$ be a binary relation over a domain (of nodes) $N$. Then $R$ is a continuous-image graph, short CIG, if it carries the *image interval* property: there is a total order $<_i$ on $N$ with the induced sequence $S$ over $N$ such that for all nodes $n \in N$, $R(n) = \varnothing$ or $R(n) = \{S[s], \ldots, S[e] : s \leq e \in \mathbb{N}\}$.

The definition of continuous image graphs allows graphs where some or all children of two parents are "shared" (in contrast to trees where this is never allowed). However, it limits the degree of sharing: Figure 41 shows two minimal graphs that are *not* CIGs. Incidentally, both graphs are acyclic and, if we take away any one edge in either graph, the resulting graph becomes a CIG. The second graph is actually the smallest (w.r.t. the number of nodes and edges) graph that is not a CIG. The first is only edge minimal but illustrates an easy to grasp sufficient but not necessary condition for violating the image interval property: if a node has at least three parents and each of the parents has at least one (other) child not shared by the others then the graph can not be a CIG.

On continuous-image graphs we can exploit similar techniques for compacting structural relations as on trees, most notably representing the nodes related to a given node as a single, continuous interval and thus with constant space. This applies also for derived relations such as closure (XPath's descendant) or order relations (XPath's following-sibling) on CIGs.

*Testing for CIGs: consecutive ones property*

Moreover, whether a given graph is a CIG (and in what order its node must be sorted to arrive at continuous intervals for each parent's children) is just another way of saying that its adjacency matrix carries the *consecutive ones* property [97]. For the consecutive-ones problem [40] gives the first linear time (in the size of the matrix) algorithm based on so called PQ-trees, a compact representation for permutations of rows in a matrix. More

recent refinements in [120] and [130] show that simpler algorithms, e.g., based on the PC-tree [129], can be achieved. We *adapt* these algorithms to obtain a linear time (in the size of the adjacency matrix) algorithm for deciding whether a graph is a CIG and computing a CIG-order.

*Practical CIGs*

From a practical perspective, CIGs are actually quite common, in particular, where time-related or hierarchical data is involved: If relations, e.g., between Germany and kings, are time-related, it is quite likely that there will be some overlapping, e.g., for periods where two persons were king of Germany at the same time. Similarly, hierarchical data often has some limited anomalies that make a modelling as strict tree data impossible. Figure 42 shows actual data[3] on relations between the family (red nodes, non-ruling member (1), co-emperor or heir designate [10], emperors (2)) of the Roman emperors in the time of the "Five Good Emperors" (Edward Gibbon) in the 2nd century. It also shows, for actual emperors, which of the four new provinces ((I)) added to the roman empire in this period each emperor ruled (the other provinces remained mostly unchanged and are therefore omitted). Arrows between family members indicate, natural or adoptive, fathership[4]. Arrows between emperors and provinces show rulership, different colors are used to distinguish different emperors. Despite the rather complicated shape of the relations (they are obviously not tree-shaped and there is considerable overlapping, in particular w.r.t. province rulership).

The previous example also highlights the intuition behind continuous-image graphs: we allow some overlapping between among the children of different nodes, but only in such a way that the images can still be represented (over some order on the nodes) as continuous intervals. Figure 43 illustrates the intervals on the Roman provinces for representing the ruled provinces of each emperor: With the given order on the provinces, each image is a single interval (e.g., Trajan I–III and Septimus Severus II–IV) even though the data is clearly not tree-shaped (or a closure relation of a tree-shaped relation).

How continuous-image graphs differ from tree-shaped data (or closure relations over tree-shaped data) is further detailed in Figure 44: Tree data carries the image disjointness property as, under the order on the nodes induced by a breadth-first traversal, the nodes in the image of any parent node in the tree form a continuous, non-overlapping interval. Closure

3 The name and status of the province between the wall of Hadrian and the wall of Antonius Pius in northern Britain is controversial. For simplicity, we refer to it as "Caledonia", though that actually denotes all land north of Hadrian's wall.

4 Note that all emperors of the Nervan-Antonian dynasty except Nerva and Commodus were adopted by their predecessor and are therefore often referred to as "Adoptive Emperors".

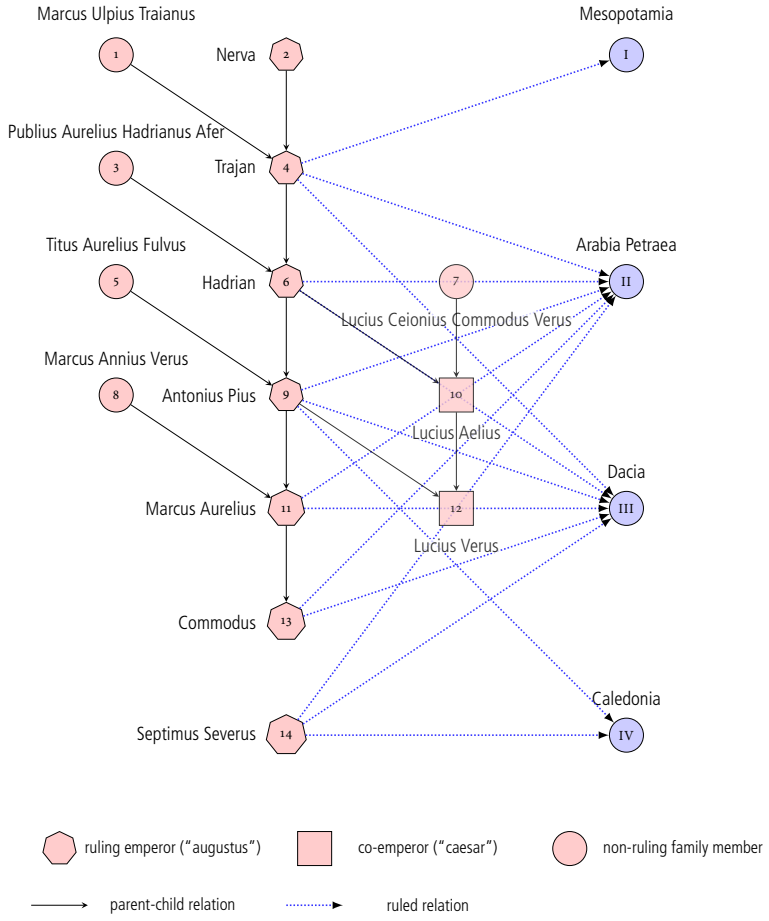



Figure 42. "The Five Good Emperors" (after Edward Gibbon), their relations, and provinces.

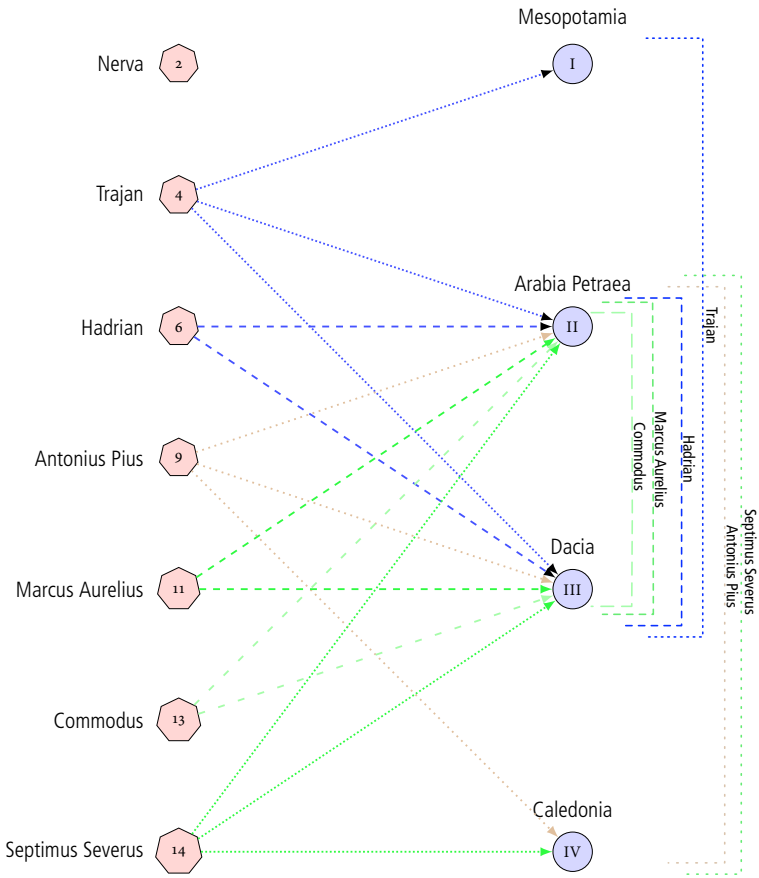



Figure 43. Overlapping of province children in the "The Five Good Emperors" example, Figure 40

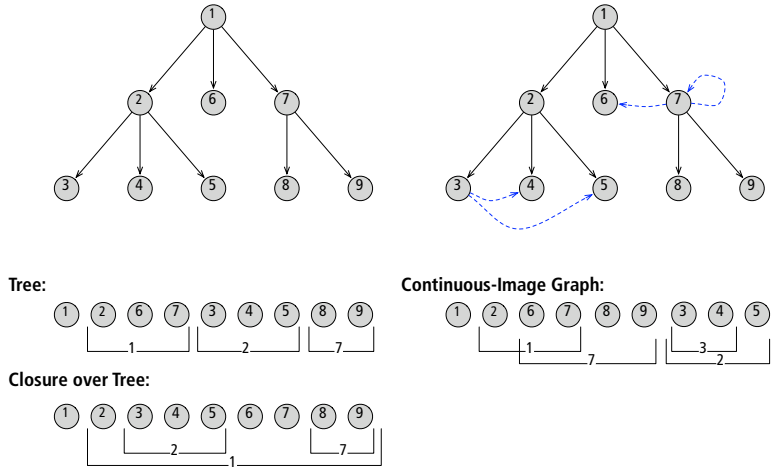



Figure 44. Overlapping of images in trees, closure relations over trees, and continuous-image graphs

relations over tree data (i.e., relations such as XPath's descendant) carry the image containment property as, e.g., under the order on the nodes induced by a depth-first traversal, again the nodes in the image of any parent node form a continuous interval and overlapping is limited: either two such intervals do not overlap at all or one is contained within the other.

Continuous-image graphs (as shown in the right of Figure 44) carry, as stated above, the image interval property, i.e., there is some order on the nodes such that the nodes in the image of each parent form a continuous interval. Here, the intervals may overlap arbitrarily as illustrated in Figure 44. However, in contrast to the tree or closure relation over tree case the required order on the nodes is no longer known a-priori but must be determined for each graph using, e.g., the above described algorithms.

## 10.3 SEQUENCE MAP: STRUCTURE-AWARE STORAGE OF TREE CORE RESULTS

*Locality of dependencies in tree queries*

When we evaluate tree queries, we can observe that for determining matches for a given query node only the match for its parent and child in the query tree are relevant. Intuitively, this "locality" property holds as in a tree there is at most one path between two nodes. To illustrate, consider, e.g., the XPath query //a//b//c selecting c descendants of b descendants of a's. Say there are $n$ a's in the data nested into each other with $m$ b's nested

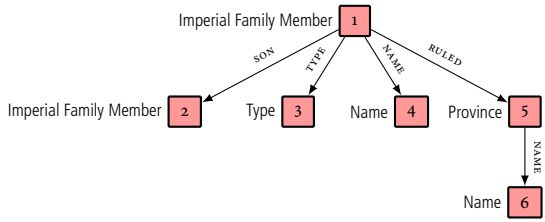



Figure 45. Selecting sons, type, name, and ruled provinces for all members of the imperial family in the data of Figure 42.

inside the a's and finally inside the b's (again nested in each other) $l$ c's. Then a naive evaluation of the above query considers all triples $(a, b, c)$ in the data, i.e., $n \times m \times l$ triples. However, whether a c is a descendant of a b is independent of whether a b is a descendant of an a. If a b is a descendant of several a's makes no difference for determining its c descendants. It suffices to determine in at most $n \times m$ time and space all b's that are descendants of a, followed by a separate determination of all c's that are descendants of such b's in at most $m \times l$ time and space.

*Multivalue dependencies*

Indeed, if we consider the answer relation for a tree query, i.e., the relation with the complete bindings as rows and the query's nodes as columns, this relation always exhibits multivalue dependencies [91]: We can normalize or decompose such a relation for a query with $n$ nodes into $n - 1$ separate relations that together faithfully represent the original relation (and from which the original relation can be reconstructed using $n-1$ joins). This allows us to compact an otherwise potentially exponential answer (in the data size) into a polynomial representation.

*Full decomposition like column store*

This is the first principle of the **sequence map**: decompose the query into separate binding sequences for each query node with "links" or pointers relating bindings of different nodes. We thus obtain an exponentially more succinct data structure for (intermediary) answers of tree queries than if using standard (flat) relational algebra. In this sense, a sequence map can be considered a fully decomposed *column store* for the answer relation.

| Imp-ID | Type | Name | Son-ID | Ruled-ID | Ruled-Name |
|---|---|---|---|---|---|
| 1 | non-ruling | Marcus Ulpius Traianus | 4 | – | – |
| 2 | augustus | Nerva | 4 | – | – |
| 3 | non-ruling | P. Aurelius Hadrianus Afer | 6 | – | – |
| 4 | augustus | Trajan | 6 | I | Mesopotamia |
| 4 | augustus | Trajan | 6 | II | Arabia Petraea |
| 4 | augustus | Trajan | 6 | III | Dacia |
| 5 | non-ruling | Titus Aurelius Fulvus | 9 | – | – |
| 6 | augustus | Hadrian | 9 | II | Arabia Petraea |
| 6 | augustus | Hadrian | 10 | II | Arabia Petraea |
| 6 | augustus | Hadrian | 9 | III | Dacia |
| 6 | augustus | Hadrian | 10 | III | Dacia |
| 7 | non-ruling | L. Ceionius Commodus Verus | 10 | – | – |
| 8 | non-ruling | M. Annius Verus | 11 | – | – |
| 9 | augustus | Antonius Pius | 11 | II | Arabia Petraea |
| 9 | augustus | Antonius Pius | 12 | II | Arabia Petraea |
| 9 | augustus | Antonius Pius | 11 | Ili | Dacia |
| 9 | augustus | Antonius Pius | 12 | III | Dacia |
| 9 | augustus | Antonius Pius | 11 | IV | Caledonia |
| 9 | augustus | Antonius Pius | 12 | IV | Caledonia |
| 10 | caesar | Lucius Aelius | 12 | – | – |
| 11 | augustus | Marcus Aurelius | 13 | II | Arabia Petraea |
| 11 | augustus | Marcus Aurelius | 13 | III | Dacia |
| 12 | caesar | Lucius Verus | – | – | – |
| 13 | augustus | Commodus | – | II | Arabia Petraea |
| 13 | augustus | Commodus | – | III | Dacia |
| 14 | augustus | Septimus Severus | – | II | Arabia |
| 14 | augustus | Septimus Severus | – | III | Arabia |
| 14 | augustus | Septimus Severus | – | IV | Caledonia |

Figure 46. Answers for query from Figure 45, single, flat relation.

| Imp-ID | Type | Name |
|---|---|---|
| 1 | non-ruling | Marcus Ulpius Traianus |
| 2 | augustus | Nerva |
| 3 | non-ruling | P. Aurelius Hadrianus Afer |
| 4 | augustus | Trajan |
| 5 | non-ruling | Titus Aurelius Fulvus |
| 6 | augustus | Hadrian |
| 7 | non-ruling | L. Ceionius Commodus Verus |
| 8 | non-ruling | M. Annius Verus |
| 9 | augustus | Antonius Pius |
| 10 | caesar | Lucius Aelius |
| 11 | augustus | Marcus Aurelius |
| 12 | caesar | Lucius Verus |
| 13 | augustus | Commodus |
| 14 | augustus | Septimus Severus |

| Imp-ID | Son-ID |
|---|---|
| 1 | 4 |
| 2 | 4 |
| 3 | 6 |
| 4 | 6 |
| 5 | 9 |
| 6 | 9 |
| 6 | 10 |
| 7 | 10 |
| 8 | 11 |
| 9 | 11 |
| 9 | 12 |
| 10 | 12 |
| 11 | 13 |

| Imp-ID | Prov-ID |
|---|---|
| 4 | I |
| 4 | II |
| 4 | III |
| 6 | II |
| 6 | III |
| 9 | II |
| 9 | III |
| 9 | IV |
| 13 | II |
| 13 | III |
| 14 | II |
| 14 | III |
| 14 | IV |

| Prov-ID | Name |
|---|---|
| I | Mesopotamia |
| II | Arabia Petraea |
| III | Dacia |
| IV | Caledonia |

Figure 47. Answers for query from Figure 45, multiple relations, normalized, no multivalue dependencies.

*Example decomposition*

To illustrate this, consider the query in Figure 45 on the data of Figure 42. The query selects sons and ruled provinces of members of the imperial family. We also record type and name of the family member and name of the province to easier talk about the retrieved data. The answers for such a query, if expressed, e.g., in relational algebra or any language using standard, flat relations to represent $n$-ary answers, against the data from Figure 42 yields the flat relation represented in Figure 46. As argued above, we can detect multivalue dependencies and thus redundancies, e.g., from emperor to province, from province to province name, from emperor (Imp-ID) to type and name.

To avoid these redundancies, we first decompose or normalize this relation along the multivalue dependencies as in Figure 47. For the sequence map, we use always a full decomposition, i.e., we would also partition type and name into separate tables as in a column store.

| Imp-ID | Son Range |
|---|---|
| 1 | 4 |
| 2 | 4 |
| 3 | 6 |
| 4 | 6 |
| 5 | 9 |
| 6 | 9–10 |
| 7 | 10 |
| 8 | 11 |
| 9 | 11–12 |
| 10 | 12 |
| 11 | 13 |

| Imp-ID | Prov Range |
|---|---|
| 4 | I–III |
| 6 | II–III |
| 9 | II–IV |
| 13 | II–III |
| 14 | II–IV |

Figure 48. Answers for query from Figure 45, multiple relations, interval pointers. The first table from Figure 47 remains unchanged.

### 10.3.1 SEQUENCE MAP FOR TREES AND CONTINUOUS-IMAGE GRAPHS

*Interval pointers*

Once we have partitioned the answer relation into what subsumes to only link tables as in column stores, we can observe even more regularities (and thus possibilities for compaction) if the underlying data is a tree or continuous-image graph. Look again at the data in Figure 42 and the resulting answer representation in Figure 47: Most emperors have not only ruled one of the new provinces Mesopotamia, Arabia Petraea, Dacia, and Caledonia but several. However, since the data is a continuous-image graph there is an order (indeed, the order of the province IDs if interpreted as roman numerals) on the provinces such that the provinces ruled by each emperor form a continuous interval w.r.t. that order. Thus we can actually represent the same information much more compactly using interval pointers or links as in Figure 48 where we do the same also for the father-son relation (although there is far less gain since most emperors already have only a single son).

Instead of a single relation spanning 28 rows and 6 columns (168 cells), we have thus reduced the information to $5 \cdot 2 + 11 \cdot 2 + 14 \cdot 3 = 74$ cells. This compaction increases exponentially if there are longer paths in a tree query (e.g., if the provinces would be connected to further information not related to the emperors). It increases quadratically with the increasing size of the tables, e.g., if we added the remaining $n$ provinces of the Roman empire ruled by all emperors in our data we would end up with $7 \cdot n$ additional rows of 6 columns in the first case (each of the 7 emperors in our data ruled all these provinces), but only $n \cdot 2$ additional cells when

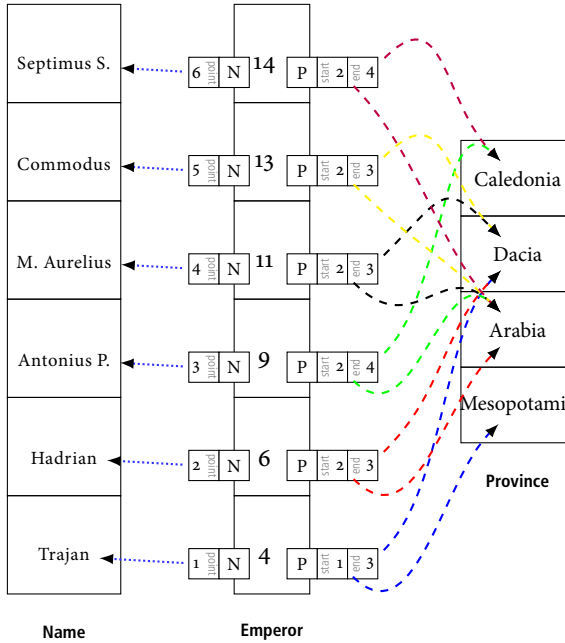



Figure 49. Sequence Map: Example. For a query selecting roman emperors together with their name and ruled provinces on the data of Figure 42.

using multiple relations and interval pointers. A detailed study of the space complexity of sequence maps is given in Section 11.4 and summarized below in Section 10.5.

Formally, we introduce the sequence map $\overset{D,Q}{\mathsf{sm}}$ on a relational structure $D$ and a query $Q$ in Chapter 11 as a mapping from a set of query nodes $V$ to sequences of matches for that query node. A *match* for query node $v$ in itself is the actual data node or edge $v$ is matched with and a set of pairs of child nodes of $v$ to start and end positions. Intuitively, it connects the match for $v$ to matches of its child nodes in the tree query. We obtain in this way a data structure as shown in Figure 49 for a query selecting roman emperors with their names and ruled provinces on the data of Figure 42.

Note, that we allow for each child node of $v$ *multiple* intervals. If the data is a CIG, it is guaranteed that only a single interval is needed and thus the overall space complexity of a sequence map is linear in the data size. However, we can also employ a sequence map for non-CIG graphs. In this case, we often still benefit from the interval pointers, but in worst-case we might need $|N \cup E|$ many interval pointers to relate to all bindings of a child variable. Overall, a sequence map for non-CIG graphs thus may use

up to quadratic space in the data size.

### 10.3.2 SEQUENCE MAPS FOR DIAMOND-FREE DAG QUERIES

Beyond tree queries, the sequence map data structure can also be used, with a slight modification, for *diamond-free DAG queries*. Diamond-free DAG queries are queries in the shape of a DAG where there are no two distinct paths between two nodes (and thus no diamond-shaped subgraph). Diamond-free DAG queries are also used in [171, 169] (there named single-join DAG queries).

*Diamond-free DAG queries: multiple order numbers per query node*

If we consider diamond-free DAG queries, there may be nodes in the query that have multiple parents. However, as for tree queries only the parent and child nodes are relevant to decide whether a data item is a match for a query node. The multiple parents, however, may be connected using different relations, e.g., XPath's child, descendant, and following relations. Above, we only demand that for each relation the continuous-image property of the data graph holds. If we have DAG queries this might lead to different, possibly incompatible orders for the image relations (e.g., child needs a breadth-first order to obtain continuous-images vs. depth-first order for descendant). However, we do not need to "strengthen" the CIG property for diamond-free DAG queries. Rather, a node with multiple parents carries a separate "order" number for each different incoming relation with incompatible orders and interval pointers are resolved as range queries over the appropriate order number. Though this increases the space need this is justified by the accompanying increase in query size. For details, see Section 10.3.2.

### 10.3.3 REPRESENTING INTERMEDIARY RESULTS: A COMPARISON

*Data structure for intermediary results*

As stated above, the sequence map is heavily influenced by previous data structures for representing intermediary answers of tree queries. Figure 50 shows the most relevant influences. Complexity and supported data shapes are compared in Section 10.5 below after discussing the actual evaluation of tree queries using the sequence map data structure. Here, we illustrate that the above discussed choices when designing a data structure for intermediary answers of tree queries are actually present in many related systems: We can find systems such as Xcerpt 1.0 [187], many early XPath processors (according to [112]), and tree algebras such as TAX [135] that use exponential size for storing all combinations of matches for each query node explicitly. [112] shows that XPath queries can in fact be evaluated in polynomial time and space, which is independently verified in SPEX [168],

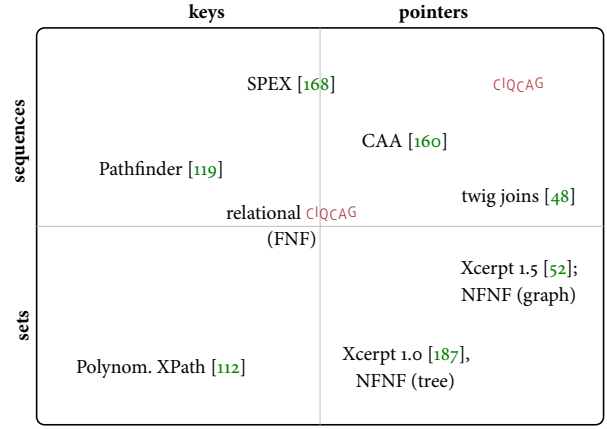



Figure 50. Data structures for intermediary results (of a tree query)

the first streaming processor for navigational XPath with all structural axes. Like SPEX and ClQcAG, complete answer aggregates [160] use interval compaction for relating matches between different nodes in a tree query. CAAs are also most closely related to ClQcAG w.r.t. the decomposition of the answer relation: fully decomposed without multivalue dependencies. In contrast, Pathfinder [39] uses standard relational algebra but for the evaluation of structural joins a novel staircase join [118] is employed that exploits the same interval principles used in CAAs and ClQcAG.

*Streams and skipping or pruning*

Streaming or cursor-based approaches such as twig join approaches [48, 70] consider the data in a certain order rather than all at once. In such a model, it is possible and desirable to skip irrelevant portions of the input stream (or relations) and to prune partial answers as soon as it is clear that we can not complete such answers. Recent versions of SPEX [57, 168] contain as most twig join approaches, mechanisms to skip over parts of the stream (at least for some query nodes) if there can not be a match (e.g., because there is no match for a parent node and we know that matches for parent nodes must come before matches for child nodes). Both twig joins and SPEX also prune results as soon as possible. However, twig joins are limited to vertical relations (child and descendant) whereas SPEX and ClQcAG can evaluate all XPath axes, though only on tree data. In Section 12.9 we discuss briefly a cursor interface for the ClQcAG algebra that iterates over the basic relations in order (of storage or CIG property). In general graphs, however, skipping and pruning is impossible, since we can never be sure that there are no more related matches. The CIG property is also too weak to ensure that we can skip and prune as match as on tree

data. However, in Section 12.9.1 we give a condition on the CIG-order of parent and child nodes in a tree query that, if it holds, ensures that these orders are "compatible" enough to allow as much skipping and pruning as in twig join approaches. This condition is more general than basic tree data, but more restrictive than the CIG property discussed above.

To summarize, though ClQcAG's sequence map is similar in its principles to several of the related approaches in Figure 50, it combines *efficient intermediary answer storage* as in CAAs with *fully algebraic* processing as in Pathfinder and *efficient skipping and pruning* as in twig joins.

Furthermore, where most of the related approaches are limited to tree data (with the notable exception of Xcerpt), ClQcAG allows processing of *many graphs*, viz. CIGs, as efficient as previous approaches allow for trees.

The sequence map data structure is exploited in the ClQcAG algebra for processing both tree and arbitrary graph queries. ClQcAG takes advantage of sequence maps to store intermediary results for the entire or for the tree core of a query as described in the following section.

## 10.4 QUERIES BEYOND TREES: GRAPHS WITH TREE CORE

The sequence map enables us to store (intermediary) answers to tree queries efficiently, in particular if the data is CIG-shaped. However, what if the query is not of tree shape?

Our answer to that question is the ClQcAG algebra. Instead of treating only tree queries (like the above mentioned approaches for XML) or entirely dropping the advantages tree queries offers (as standard relational algebra does), ClQcAG separates query processing in two phases: in the first phase we exploit *efficient algorithms for tree queries* evaluating a *tree core* of the original query. Any remaining parts of the query, if any, are evaluated on top of the resulting sequence map from the tree core evaluation.

*Principles of ClQcAG*

ClQcAG is **(1)** *designed around the sequence map* data structure and, if used in conjunction with a sequence map, achieves the same or better time and space complexity for CIG data graphs as the best known approaches for tree data. **(2)** *immediately familiar* to anyone with knowledge on the standard relational algebra. In fact, the semantics of all ClQcAG operators is purely relational (can be expressed in terms of standard relational algebra[5]). Using the sequence map and its associated algorithms provides an *equivalent yet more efficient* implementation of the ClQcAG operators.

5 With value invention, grouping, and ordering, if construction of new graphs is also considered.

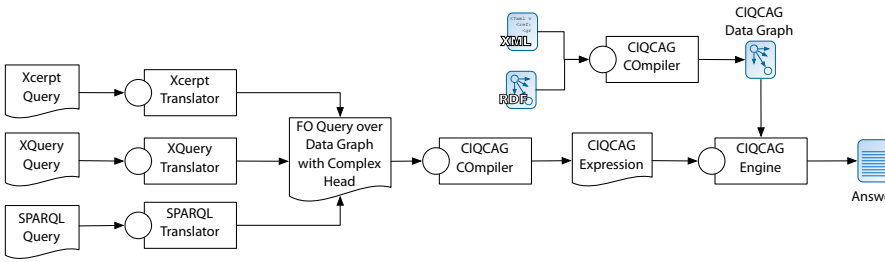



Figure 51. CIQCAG Architecture

**(3)** *able to process tree and graph queries* efficiently by means of a three-stage evaluation: in the *map construction* stage a tree core of the query is computed and specific operators allow the efficient evaluation of that tree core. In the *map expansion* stage the remaining non-tree relations are considered. Finally, there is a set of *result construction* operators that allow the construction of new data graphs (where the previous two stages operate exclusively on the input data). **(4)** for most of the presentation, limited to composition-free queries (in the sense of [144]), but in Section 13.4 we discuss the addition of iteration and recursion on top of the three stage processing described here. **(5)** able to be combined with *most approaches for indexing Web* data such as reachability indices [116, 195] or path indices [107, 71] as well as other "computed" or *derived relations* such as computed closure relations, complex binary path expressions [31], etc. This allows a great flexibility in choosing how a relation is actually realized: extensional, using some index or labeling scheme, computed ad-hoc, etc. **(6)** *accompanied by a rich set of algebraic laws*, many familiar from relational algebra, but with some additions to handle new operators and the specificity of the multi-stage evaluation.

*Realization*

Based on these principles, however, there are still a number of trade-offs when realizing the CIQCAG algebra, cf. Section 12.1 for details. Most are related to how the "links" between the decomposed relations in a sequence map are realized. In the purely relational variant we use table with order numbers and range queries. Though theoretically only slightly more costly than the other variants ($\mathcal{O}(\log n)$ instead of $\mathcal{O}(1)$ for deciding whether two matches for parent-child query nodes are related), practical performance of range queries is often disappointing in current relational databases (cf. [119]). In [52], we proposed a representation of the links by nesting (and sharing) of matches for child nodes within parent nodes (similar to nested

relational data but with sharing to avoid duplication). Though this yields better lookup time, we pay with quadratic space in worst-case even for tree and CIG data. The third variant uses ordered relations (like SQL tables) and random access to rows in these ordered relations. This allows for constant lookup time and (in case of tree or CIG data) linear space, but comes at the cost of slightly more involved reorganization, cf. Section 12.

*CIQCAG for actual Web query languages*

The above principles also make CIQCAG eminently suitable for implementing practical Web query languages. Indeed, in Part III, we show how to translate (large subsets of) XQuery, Xcerpt, and SPARQL into CIQLOG, a high-level calculus for CIQCAG, which is then translated into CIQCAG expressions as detailed in Section 13 and illustrated in Figure 51. CIQCAG is flexible enough to evaluate programs in any of these languages and to actually mix them within the same query. It is worth noting, that both the translation from (composition-free) XQuery to CIQLOG and the equivalence of calculus and algebra are accompanied by formal proofs. CIQCAG is, thus, one of the few *formally* correct algebras for (composition-free) XQuery.

### 10.4.1 OPERATOR OVERVIEW

*Overview of CIQCAG operators*

The main operators of the CIQCAG algebra are immediately familiar from relational algebra. However, CIQCAG operator's differ in a few noteworthy points (the full details as well as the operators' implementation on top of the sequence map are explained in Chapter 12): **(1)** There are *three sets of operators*, as shown in Table 50, one for each stage of the query evaluation. Mostly identical to their relational counterpart are the operators for *map expansion* (renaming $\rho$, join $\bowtie$, union $\cup$, difference $-$, projection $\pi$, and duplicate elimination $\delta$). However, we add an operator for accessing specific columns from a sequence map computed in the previous stage as flat relation ($\mathcal{F}$). **(2)** For *map construction*, we also introduce a new access operator $\mu$ that creates a sequence map from a given relation $R$. The other operators closely mirror relational operators and thus the operators of the map expansion phase, but there is one more addition: the propagation operator. **(3)** When we restrict the values for a given column in a classical, flat relation (e.g., by selection or join), we can omit tuples with values violating the selection or join criterion in a single pass over the table. However, the sequence maps store the (so far) computed answers like a column store. Though this allows more efficient storage in face of multivalue dependencies as in tree queries, eliminating values in a single column no longer immediately affects other columns. Rather when a value is eliminated in one column we have to *propagate* this fact to all related columns (and from there on recursively). CIQCAG provides for this

| access | join (conjunction) | union | difference |
|---|---|---|---|
| $\ddot{\mu}_{v,v'}(D,Q), \ddot{\mu}_{v}(D,Q,R)$ | $\ddot{\bowtie}^{(\lightning)}_{\cap}, \ddot{\bowtie}^{(\lightning)}, \ddot{\ltimes}^{(\lightning)}(S,S')$ | $\ddot{\cup}(S,S')$ | $\ddot{\setminus}(S,S')$ |
| **projection, rename** | **selection** | **propagation** | **expansion** |
| $\ddot{\pi}_V(S), \ddot{\rho}_{v_1 \to v_2}(S)$ | $\ddot{\sigma}^{(\lightning)}_c(S)$ | $\ddot{\omega}^{\uparrow}_v(S), \ddot{\omega}^{v}_{\downarrow}(S)$ | $\mathcal{F}_V(S)$ |

Table 50. Overview of sequence map operators in CIQCAG (all operators return a single sequence map $S$ except $\mathcal{F}$ which returns a (standard) relation)

propagation an explicit operator, the up- and down propagation operators $\ddot{\omega}^{\uparrow}$ and $\ddot{\omega}_{\downarrow}$. This allows us to perform several value restrictions on the same column before propagation all eliminated values at once, rather than propagation after each restriction. The price is that a sequence map may, after a restriction and before the corresponding propagation, be in an inconsistent state, where values in one column link to no longer valid values of another one. However, this price is clearly offset by the gain in complexity and performance: Implicit propagation at each restriction raises the worst-case complexity of query evaluation by a *multiplicative* factor $q$ (query size). Using explicit separate propagation, it suffices to propagate each column's values to its parent columns once at the end of the map construction leading to an *additive* factor of $q \times n$. This is ensured by the translation of CIQLog to CIQCAG (to ensure correct answers) as described in Chapter 13. For more details on explicit propagation see Section 12. **(4)** Finally, result construction is similar to other complex value algebras (cf. [119, 144] with operators for value invention $\nu$, ordering ($\omega$), conditional construction, and graph construction.

*Query example*

To illustrate, how these operators play together to implement a typical Web query consider the query in Figure 52 (4 indicates a answer node, all other nodes are existentially quantified; $\neq_{id}$ indicates a anti-join based on node identity). This is a graph query, but a spanning tree covers almost the entire query. The query is realized in CIQCAG by the following expression (we decide to treat the SON edge from 2 to 4 as only non-tree edge[6]):

**6** The tree core of a query is not unique at all: We could also have chosen to consider also the $\neq_{id}$ as non-tree or to duplicate node 4 and introduce a identity join between the original and the duplicated node in the map expansion phase.

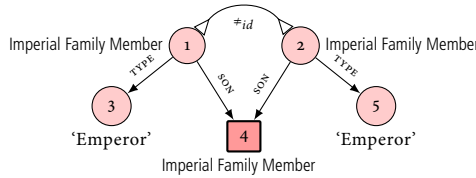



Figure 52. Selecting all sons of Roman emperors that have a double claim to the throne (two (distinct) fathers that where both emperors).

$$F_{v_2,v_4}\Big(\ (\dddot{\mu}_{v_1,v_4}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\pi}_{v_1}(\dddot{\mu}_{v_1,v_3}(D,Q) \ddot{\bowtie}_{\cap} \dddot{\mu}_{v_3}(D,Q,\text{'Emperor'})))$$
$$\ddot{\bowtie}_{\cap} \dddot{\mu}_{v_1,v_2}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\pi}_{v_2}(\dddot{\mu}_{v_2,v_5}(D,Q) \ddot{\bowtie}_{\cap} \dddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\ \Big)$$

Within the flatten operator, we find the expression responsible for evaluating the tree part of the query: we join sequence maps for type and son edges as well as label restrictions for nodes 3 and 5. Since we are not interested in the actual matches for 3 and 5 we use semi-joins instead of full joins. In the non-tree part we grab $v_2$ and $v_4$ from the sequence map ($v_1$ is not needed any more) and join them with the SON relation between $v_2$ and $v_4$. The result is projected to $v_4$ as this is the only answer node.

### 10.4.2 TREE CORES AND HYPERTREES

*Tree cores—how to compute them*

As seen in the previous example, we have considerable leeway how to choose the tree core of a graph query. We discuss choosing a good tree core very briefly in Chapter 13. In addition to usual considerations about selectivity and size of the involved relations, we have in our case an additional guide: if at all possible the chosen relations should carry the CIG property, i.e., exhibit an order on the data nodes such that the image of a node forms a continuous interval.

*Hypertree decompositions*

For relational queries, it has been shown that acyclic or tree queries are not the largest class of polynomial time queries. Rather queries with bounded tree- or hypertree-width [108] can still be evaluated in polynomial time. Nevertheless, we choose a tree decomposition. Hypertree decompositions yield groups of query nodes (variables) that are strongly connected in a query, but loosely connected with the rest of the query. If there is a bound on the size of these groups, we can evaluate each group separately and only expose the few connection points to the rest of the query. In contrast, for tree queries the groups are always single query nodes. For our purpose, however, hypertree decompositions seem less attractive

than for standard relational queries for two reasons: **(1)** to treat multiple incoming edges (even if there number is bound by some constant) the sequence map data structure needs to be extended considerably and **(2)** the CIG property also needs to be adapted to guarantee *one* order among all relations involved in a hypertree node. In addition, the integration of techniques such as path indices and twig joins that have been developed for tree queries with CIQCAG is almost free if we consider tree cores, but is less than obvious if we consider hypertree decompositions.

However, for future work a closer inspection of hypertree decompositions (and, in particular, their effect on the CIG property) would be strongly desirable, see also Chapter 15.1.

## 10.5 COMPLEXITY AND CONTRIBUTIONS

With a first intuition on the data structures and operations that form the CIQCAG algebra, we can turn to investigate some of its properties. For a detailed analysis of the space complexity of the sequence map see Section 11.4, the complexity of query evaluation with CIQCAG is analysed in Chapter 12.

*Space and time complexity of CIQCAG query evaluation*

Table 52 summarizes the complexity results for query evaluation with the CIQCAG algebra (using the sequence map as data structure). It is worth highlighting data and query space complexity are both linear for tree and CIG data and tree queries. Also note, that there is no penalty at all to going from tree data to CIG data, but arbitrary graph data does incur a linear penalty (both in space and time). The logarithmic factor in the time complexity can be discarded, if we assume that all relations can be accessed in CIG order. This is possible, since the CIG property is a static property of the data and does not depend on the query. Otherwise, we need to sort relations to add them to the sequence map. For Table 52, we assume the latter.

*Membership test*

One factor in the time complexity of CIQCAG is the cost of the membership test in a relation. For extensional relations this is constant. However, as argued above, CIQCAG is also well suited to handle derived relations such as descendant with out without index. In this cases $m$ may in fact have significant influence on the query evaluation. For the evaluation of the Web query languages discussed in Part III (Xcerpt, XQuery, SPARQL), extensional relations and structural closure relations (XPath's descendant, following, etc.) suffice. As discussed in Section 6.5.1, for tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [86] or other labeling schemes such as [200]). However, for graph data this is not so obvious. Fortunately, there has been considerable research on reachability or closure relations and

| data \ queries | path queries | tree queries | graph queries |
|---|---|---|---|
| tree | $\mathbf{T} = \mathcal{O}(q \cdot n)$ | $\mathbf{T} = \mathcal{O}(q \cdot n)$ | $\mathcal{O}(n^{q_g} + T_{\text{tree}})$ |
| | $\mathbf{S} = \mathcal{O}(q \cdot n)$ | $\mathbf{S} = \mathcal{O}(q \cdot n)$ | $\mathcal{O}(n^{q_g} + S_{\text{tree}})$ |
| CIG | $\mathbf{T} = \mathcal{O}(q \cdot n)$ | $\mathbf{T} = \mathcal{O}(q \cdot n)$ | $\mathcal{O}(n^{q_g} + T_{\text{tree}})$ |
| | $\mathbf{S} = \mathcal{O}(q \cdot n)$ | $\mathbf{S} = \mathcal{O}(q \cdot n)$ | $\mathcal{O}(n^{q_g} + S_{\text{tree}})$ |
| graph | $\mathbf{T} = \mathcal{O}(q \cdot n^2 \cdot m)$ | $\mathbf{T} = \mathcal{O}(q \cdot n^2 \cdot m)$ | $\mathcal{O}(n^{q_g} + T_{\text{tree}})$ |
| | $\mathbf{S} = \mathcal{O}(q \cdot n^2)$ | $\mathbf{S} = \mathcal{O}(q \cdot n^2)$ | $\mathcal{O}(n^{q_g} + S_{\text{tree}})$ |

Table 52. Complexity of query evaluation with CIQCAG algebra ($q$ query size, $n$ data size, $m$ complexity of membership test—assumed constant for all tree, forest, or CIG shaped relations, $q_g$: number of "graph" variables, i.e., variables with multiple incoming query edges)

their indexing in arbitrary graph data in recent years. Table 54 summarizes the most relevant approaches for our work. Theoretically, we can obtain constant time for the membership test if we store the full transitive closure matrix. However, for large graphs this is clearly infeasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test.

The first class are based on the idea of a 2-hop cover [76]: Instead of storing a full transitive closure, we allow that reachable nodes are reached via at most one other node (i.e., in two "hops"). More precisely, each node $n$ is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach $n$, $out(n)$ a set of nodes that are reachable from $n$. Both sets are assigned in such a way, that a node $m$ is reachable from $n$ iff $out(n) \cup in(m) \neq \varnothing$. Unfortunately, computing the optimal 2-hop cover is NP-hard and even improved approximation algorithms [189] have still rather high complexity.

| approach | characteristics | query time | index time | index size |
|---|---|---|---|---|
| **Shortest path** [174] | no index | $\mathcal{O}(n+e)$ | – | – |
| **Transitive closure** | full reachability matrix | $\mathcal{O}(1)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^2)$ |
| **2-Hop** [76] | 2-hop cover[a] | $\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$ | $\mathcal{O}(n^4)$ | $\mathcal{O}(n \cdot \sqrt{e})$ |
| **HOPI** [189] | 2-hop cover, improved approximation algorithm | $\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n \cdot \sqrt{e})$ |
| **Graph labeling** [5] | interval-based tree labeling and propagation of intervals of non-tree descendants. | $\mathcal{O}(n)^b$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(n^2)^c$ |
| **SSPI** [70] | interval-based tree labeling and recursive traversal of non-tree edges | $\mathcal{O}(e-n)$ | $\mathcal{O}(n+e)$ | $\mathcal{O}(n+e)$ |
| **Dual labeling** [199] | interval-based tree labeling and transitive closure over non-tree edges | $\mathcal{O}(1)^d$ | $\mathcal{O}(n+e+e_g^3)$ | $\mathcal{O}(n+e_g^2)$ |
| **GRIPP** [195] | interval-based tree labeling plus additional interval labels for edges with incoming non-tree edges | $\mathcal{O}(e-n)$ | $\mathcal{O}(n+e)$ | $\mathcal{O}(n+e)$ |

***a*** Index time for approximation algorithm in [76].

***b*** More precisely, the number of intervals per node. E.g., in a bipartite graph this can be up to $n$, but in most (sparse) graphs this is likely considerably lower than $n$.

***c*** More precisely, the total number of interval labels.

***d*** [199] introduces also a variant of dual labeling with $\mathcal{O}(\log e_g)$ query time using a, in practical cases, considerably smaller index. However, worst case index size remains unchanged.

Table 54. Cost of Membership Test for Closure Relations. $n$, $e$: number of nodes, edges in the data, $e_g$: number of non-tree edges, i.e., if $T(D)$ is a spanning tree for $D$ with edges $E_{T(D)}$, then $e_g = |E_D \setminus E_{T(D)}|$.

A different approach [5, 70, 199, 195] is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [199] where a full transitive closure over the non-tree edges is build. GRIPP [195] and SSPI [70] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

For CIQCAG we can choose any of the approaches. For the following, we assume constant time membership, since that is easily achieved on trees and feasible with approaches such as Dual Labeling even for graphs.

*Comparison with existing approaches for Web query evaluation*

With this understanding about the complexity of query evaluation with CIQCAG, we can move to compare CIQCAG with representative approaches for Web querying developed in recent years (mostly for XML data). Table 56 summarizes this comparison: It shows that CIQCAG rivals the best known approaches for tree data (twig joins and SPEX), but in contrast to those approaches extends the same complexity to CIG data and can also, albeit at a cost, handle graph queries (like structural joins used in most other XML or RDF algebras). Note, that in all cases we consider pointers of constant size (as done in [48], [168], and [160]). In fact, all approaches need an additional log $n$ multiplicative factor if pointer size is taken into consideration.

*Contributions*

To summarize, CIQCAG is a novel algebra for Web queries that

- is based on a novel data structure for *efficient storage of intermediary results* of tree queries that is exponentially more succinct than purely relational approaches.
- extends previous approaches for querying tree data to a larger class of data graphs, called *continous-image graphs*. The CIG property can be tested in polynomial time and is independent of the query.
- *rivals the best known approaches* for tree query evaluation on tree data yet extends their properties (complexity and skipping, cf. Section 12.9.1) to CIG data.
- *gracefully degrades* with the increasing "graphness" of data and queries .
- allows for *easy integration* of derived relations and indices such as interval labeling, graph reachability indices, path indices, etc.
- provides a provable correct evaluation for large, relevant fragments of practical query languages, viz. XQuery, Xcerpt, and SPARQL.

| | query type | time | space |
|---|---|---|---|
| **Structural Joins** | path queries | *path index:* | |
| decompose query; test each structural constrain individually; join the results; | | (DataGuide [107], IndexFabric [78], [71])[a] | |
| | | $\mathcal{O}(m_{path}) \sim \mathcal{O}(d)$ | $\mathcal{O}(n)$; $\mathcal{O}(d \cdot n)$ index |
| | | *no* path index, *standard join* [7, 116, 112]) | |
| | | $\mathcal{O}(n^{q_a} + q \cdot n \cdot \log n \cdot m)$[b] | $\mathcal{O}(n^{q_a} + q \cdot n^2)$ |
| | | *no* path index, *structure-aware join*, ([39]; tree data only) | |
| | | $\mathcal{O}(n^{q_a} + q \cdot n)$ | $\mathcal{O}(n^{q_a} + q \cdot n^2)$ |
| | tree queries | with path index (tree/DAG data only) | |
| | | $\mathcal{O}(n^{q_a} + b \cdot n \cdot \log n \cdot m_{path}))$ | $\mathcal{O}(n^{q_a} + b \cdot n^2)$ |
| | | *no* path index | |
| | | $\mathcal{O}(n^{q_a} + q \cdot n \cdot \log n \cdot m)$ | $\mathcal{O}(n^{q_a} + q \cdot n^2)$ |
| | graph queries | $\mathcal{O}(n^q)$ | $\mathcal{O}(n^q)$ |
| **Twig or Stack Joins** | tree data | $\mathcal{O}(q \cdot n)$[c] | $\mathcal{O}(q \cdot n + n \cdot d)$[d] |
| holistic (single operator) [48]; partial answers in $q$ stacks; parent pointers connect stack entries; | | | |
| limited to only child/descendant relations in trees and DAGs [70]; "longer" skip distance when using indices [Jiang]; | graph data | $\mathcal{O}(q \cdot n^2 + e)$[e] | $\mathcal{O}(q \cdot n + e)$[f] |
| **SPEX** (and similar streaming engines) [171, 168]; similar to twig join but with a single input stream and pointers realized as conditions; additionally supports horizontal axes; skipping added in [57]; | | $\mathcal{O}(q \cdot n^2)$[g] | $\mathcal{O}(q \cdot n \cdot d)$[h] |
| **Complete Answer Aggregates** | | $\mathcal{O}(q \cdot n \cdot \log n \cdot d)$ | $\mathcal{O}(q \cdot n \cdot d)$ |
| [160]; manages *all* answers as CIQCAG; limited to tree data and structural relations (e.g., no value joins); similar to twig join without stack management | w/o closure axes | $\mathcal{O}(q \cdot n \cdot \log n)$ | $\mathcal{O}(q \cdot n)$ |
| CIQCAG | tree queries | $\mathcal{O}(q \cdot n)$[i] | $\mathcal{O}(q \cdot n)$[i] |
| sequence map variant; skipping may reduce average case | | $\mathcal{O}(q \cdot n^2 \cdot m)$ | $\mathcal{O}(q \cdot n^2)$ |
| | graph queries | $\mathcal{O}(n^{q_g} + q \cdot n)$[i] | $\mathcal{O}(n^{q_g} + q \cdot n)$[i] |
| | | $\mathcal{O}(n^{q_g} + q \cdot n^2 \cdot m)$ | $\mathcal{O}(n^{q_g} + q \cdot n^2)$ |

***a*** limited to unary path queries with only child/descendant relations against trees

***b*** skipping reduces average case by using indices such as [116], XR-Tree [136], BIRD [200]

***c*** More precisely, $\mathcal{O}(q \cdot \max(b_i, d))$ where $b_i$ is the average of bindings per query variable. Both $b_i$ and $d$ are, in worst case, $n$. $b_i \ll n$ only if the selectivity of the node tests in the query is high.

***d*** Answers are progressively generated.

***e*** More precisely, $\mathcal{O}(q \cdot \max(b_i, d)^2)$ where $b_i$ is the average of bindings per query variable. Both $b_i$ and $d$ are, in worst case, $n$.

***f*** Answers are progressively generated.

***g*** Lower complexity for limited fragments, e.g., $\mathcal{O}(q \cdot n \cdot d)$ if only vertical axes are present.

***h*** In some fringe cases, complexity degenerates to $\mathcal{O}(q \cdot n \cdot d + n^2)$, for details see [168]. Answers are progressively generated.

***i*** For tree and continuous-image graphs.

Table 56. Comparison of Related Approaches. $n$: number of nodes in the data, $d$: depth, resp. diameter of data; $e$: number of edges; $q$: size of query, $q_a$: number of result or answer variables; $q_g$: number of "graph" variables, i.e., variables with multiple incoming query edges; $m$ maximum time complexity for relation membership test; $m_{path}$ time complexity for path index access.

*Overview*

In the following three chapters, we illustrate CIQCAG and its properties in more details. Chapter 11 presents the sequence map data structure, its formal definition, and properties (including space complexity). Chapter 12 introduces the operators of the CIQCAG algebra together with purely relational semantics, algorithms based for a realization using the sequence map data structure, and their complexity. We conclude the presentation of the algebra in Chapter 13 by showing the equivalence with the CIQLog calculus (see Chapter 6), how to compute the tree core of a graph query, and proofing a number of algebraic equivalences for optimizing CIQCAG expressions.

# 11 SEQUENCE MAP



## 11.1 INTRODUCTION

As discussed in the previous section, the sequence map data structure stands at the core of the ClQcAG algebra: It allows us the polynomial, in many cases even linear, storage of (intermediary) answers to a tree core of a query. For tree queries all evaluation is done directly on the sequence map, for graph queries we must evaluate the remaining non-tree relations separately, but still profit in most cases greatly by reducing the size of the non-tree answers.

Let us briefly recall, from the previous section, the main motivations for designing a new data structure:

**(1)** When we evaluate tree queries, we can observe that for determining matches for a given query node only the match for its parent and child in the query tree are relevant.

**(2)** Indeed, if we consider the answer relation for a tree query, i.e., the relation with the complete bindings as rows and the query's nodes as columns, this relation always exhibits multivalued dependencies [91].

**(3)** To avoid these dependencies, we fully decompose the answer relation as in a column store: binding sequences for each query node with "links" or pointers relating bindings of different nodes. This gives us an exponentially more succinct storage than a flat relation.

**(4)** Once we have partitioned the answer relation into what subsumes to only link tables as in column stores, we can observe even more regularities (and thus possibilities for compaction) if the underlying data is a tree or continuous-image graph. These regularities allow us to represent the image of a node under as a single, continuous interval and thus yield an even smaller representation of the intermediary answers.

In the remainder of this section, we illustrate how these principles and observations are exploited in the definition of the sequence map data structure to obtain a space optimal data structure for tree queries on tree, forest, and CIG data (linear space data complexity). The same data structure can also be used on arbitrary graph data where it is more compact, for most cases, than a decomposed relation without interval pointers, though its worst-case space complexity is the same. We start this illustration with the formal definition of the sequence map in Section 11.2, continue it by taking a closer look at the effect of data shape on the interval representation of related answers in sequence maps (cf. Section 11.3, and conclude with a study of the space complexity of the sequence map in Section 11.4. We also briefly glance at some variations of the basic sequence map that allow us to cover a slightly larger fragment of queries albeit at a slight increase in the time complexity of most operations on the sequence map (which are discussed in the next chapter).

## 11.2 SEQUENCE MAP: A DATA-STRUCTURE FOR THE DECOMPOSED REPRESENTATION OF INTERMEDIARY ANSWERS TO TREE QUERIES

To hold intermediary results of tree cores of queries, we define a compact data structure, called SEQUENCE MAP. As sketched above, this data structure holds (intermediary) results of $n$-ary tree queries while avoiding the multivalued dependencies that occur if flat relations are employed for this purpose. By avoiding these dependencies and storing the results fully

decomposed, we obtain an exponentially more succinct data structure than a flat relation.

Additionally, we exploit the properties of the queried data: Where the data permits (as in the case of trees and CIGs), we compact the pointers (sometimes also called "links", references, or foreign keys) between the decomposed representations for bindings of different variables into intervals. For this, we store bindings of each variable as a single sequence with interval points to related child variables associated to each binding.

*Data: binary relational structures*

Formally, we define a sequence map over the queried data and the (tree) query to be evaluated. As data, we consider a (slightly extended)[1] *relational structure* $D$. $D$ is defined over a relational schema $\Sigma = (R_1[U_1], \ldots, R_k[U_k])$ and a *finite* domain $N$ of nodes (or objects or elements or records) in the data. Each $R_i[U_i]$ is a relation schema consisting in a relation name and a nonempty set of attribute names. We assume an equality relation $=$ on the nodes that relates each node to itself only (*identity*). $D$ is a tuple $(R_1^D, \ldots, R_k^D, O)$. Each $R_i^D$ is a finite, unary or binary relation over $N$ with name $R_i$. For a relation $R$, $\mathsf{ar}(R)$ denotes its arity. We extend $D$ with an order mapping $O$ that associates with each (binary) $R_i$ a total order on $N$ such that all $n \in \operatorname{rng} R_i$ are before all $n' \in N \setminus \operatorname{rng} R_i$. We denote with $O(D) = \{o : \exists R_i \in D : O(R_i) = o\}$ the set of orders to which the relations in $D$ are mapped. These orders serve to represent the image of each node in a relation as one or more continuous intervals over the order associated with that relation. Choosing an appropriate order for a relation is discussed in Sections 11.3.1 (for tree data) and 11.3.2 (for CIG data).

*Derived relations*

Note, that we do not assume that all relations in $D$ are extensional. Rather some might be derived (e.g., as closure) from other relations. See Chapter 5 for a discussion on relations to represent XML and RDF documents and how to use these relations to translate XQuery and Xcerpt queries into CloCaG expressions.

*Queries: trees*

For the example data in Figure 42, appropriate relations are the son-relation between members of the imperial family, the ruled-relation between emperors and (newly constituted) provinces. For both relation the node IDs in Figure 42 give a suitable associated order that allows the representation of the relation with a single interval pointer per parent node as described in Section 11.3.1. We use unary relations to classify nodes by type. In this case, we use type relations for imperial family members, emperors, co-emperors, and provinces each. Furthermore, we explicitly represent names of provinces and family members attached by a name relation.

1 The deviation lies in the addition of order for each relation. Furthermore, we restrict ourselves to binary relations.

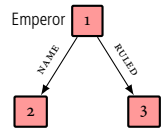



Figure 53. Selecting all Roman emperors together with their name and ruled provinces.

A sequence map is used to store intermediate results for one specific tree query. A tree query, in our context, is any tree over a set of variables (or query nodes or attributes of the result relation). Each node is a query variable, each edge is labeled with a relation that must hold between bindings for the query variables connected by the edge. Formally, a query $Q$, over a relational schema $\Sigma$ using the relation names $\mathcal{R}_1$ for unary relations and the relation names $\mathcal{R}_2$ for binary relations, is a tree[2] $(V, \mathcal{L}_V, E = V \times V, r, \mathcal{L}_E)$. The set of variables $V$ serve as nodes of the tree with the root $r$ and are labeled with (sets of) relation names by $\mathcal{L}_V : V \to 2^{\mathcal{R}_1}$. Edges in $E$ connect pairs of variables and are labeled with the labeling function $\mathcal{L} : E \to \mathcal{R}_2$ connects each edge in $Q$ with a relation name.

**Definition 11.1** (Children, Parent, and Relation in Queries)**.** For each variable (or node) $v$ in $Q$, we denote with

**(1)** $\mathsf{children}(v) = \{v' \in V : (v, v') \in E\}$ the child variables of $v$;

**(2)** $\mathsf{parent}(v) = v'$ where $(v', v) \in E$, the parent variable of $v$ or $\perp$ if $v$ is the root.

**(3)** $\mathsf{rel}(v) = \mathcal{L}((\mathsf{parent}(v), v))$ the name of the relation between the parent variable and $v$ or $\perp$ if $v$ is the root.

In the following, we assume that all relations used in a query $Q$ are also defined in the relational structure $D$ the query is evaluated against.

Figure 53 illustrates an example query tree with the nodes $\{1, 2, 3\}$, the edges $\{(1, 2), (1, 3)\}$, the root 1, the edge labeling $\{(1, 2) \to \text{RULED}, (1, 3) \to \text{NAME}\}$, and the node labeling $\{1 \to \text{Emperor}\}$.

*Sequences*

Bindings for variables are stored in the sequence map fully decomposed, one sequence per variable.

**Definition 11.2** (Sequence)**.** A SEQUENCE $S$ from some (finite) set $N$ is a finite sequence on $N$, or more formally a bijective function from

2 In fact, sequence maps can be used for forest an diamond-free DAG queries, see Section 11.5.2. For clarity of presentation, we limit ourselves here to tree queries.

$\{1, \ldots, |N|\}$ to $N$. For a sequence $S$, the $i^{\text{th}}$ element of the sequence is $S(i)$ and is sometimes denoted as $S[i]$ to emphasize that $S$ is a sequence. $i$ is referred to as *index* of $S[i]$ in $S$.

Note, that we allow only *duplicate free* sequences in this definition, i.e., a $n \in N$ occurs at most once in a sequence $S$. Therefore, we can denote with $S^{-1}(n)$ the index of $n$ in $S$.

For a finite set $N$, we define the *set of all subsequences*, denoted $SubSeq(N)$, as the set $\{S : \{1, \ldots, k\} \to N' : N' \subset N \wedge k = |N'| \wedge S \text{ bijective}\}$.

We call a sequence $S \in SubSeq(N)$ *consistent* with an order $<_N$ over $N$ if for all $n_1, n_2 \in N$ it holds that, if there are $i, j$ with $S[i] = n_1$ and $S[j] = n_2$, then $n_1 <_N n_2$ implies $i < j$. If $<_N$ is total, $n_1 <_N n_2$ iff $i < j$.

Finally, any total order on $N$ is naturally represented by a sequence over $N$ and vice versa:

**Definition 11.3** (Induced sequence)**.** Let $<$ be a *total* order over some domain $N$. Then $S_<$ is called the induced sequence for $<$ over $N$ if $S_<[i] = n \in N$ with $|\{n' \in N : n' < n\}| = i$. $S_<$ is, by definition, *consistent* with $<$.

Obviously, computing the induced sequence is the same as sorting $N$ w.r.t. $<$ and thus has $\Omega(n \log n)$ time complexity.

*Sequence Map: formal definition*

**Definition 11.4** (Sequence Map)**.** A SEQUENCE MAP $\overset{D,Q}{\mathsf{sm}}$ on an (extended) relational structure $D$ and a (tree) query $Q$ over $D$ is a mapping from the set of variables $\mathsf{Vars}(Q)$ to sequences of bindings for these variables to nodes in $D$. For each variable binding, we also record a set of intervals of related bindings for each child variable. This way, a binding $b$ for a variable $v$ (at index $i$) is associated with a set of triples $(v', s, e)$ that indicates that all bindings $b'$ with index $s \leq j \leq e$ for the child variable $v' \in \mathsf{children}(v)$ are related to $b$.

Let $Intervals = \{(i, j) \in \mathbb{N}^2 : i \leq j\}$ be the set of all intervals of integers. Then, we obtain the following signature for a sequence map:

$$\overset{D,Q}{\mathsf{sm}} : \mathsf{Vars}(Q) \to SubSeq(\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q) \times Intervals})$$

Note, that in each sub-sequence each $n \in \mathsf{Nodes}(D)$ occurs at most once and is associated to a single set of pairs of variables and intervals.

For any $v \in V$, we write $\overset{D,Q}{\mathsf{sm}}(v)$ to indicate the sequence of bindings for $v$. With $\mathsf{binding}(\overset{D,Q}{\mathsf{sm}}(v)[i])$ we denote the actual binding node in the $i$-th entry for $v$ (or some distinct value $\perp$ with $\perp > n$ for all $n \in \mathsf{Nodes}(D)$ if $i > |\overset{D,Q}{\mathsf{sm}}(v)|$), with $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathsf{sm}}(v)[i])$ the set of intervals associated with $v'$ in the $i$-the entry for $v$ in the sequence map (or $\varnothing$ if $i > |\overset{D,Q}{\mathsf{sm}}(v)|$). Finally, we write $\mathsf{Nodes}_{v'}(\overset{D,Q}{\mathsf{sm}}(v)[i])$ to indicate the set of bindings for $v'$ covered by an interval in $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathsf{sm}}(v)[i])$.

In addition to the above signature, we place three further restrictions on any sequence map $\overset{D,Q}{\mathbf{sm}}$:

**(1)** For each variable $v$ and index $i$, the set of intervals $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathbf{sm}}(v)[i])$ is *non-overlapping*. A set $M$ of intervals is non-overlapping if, for each pair of intervals $(s_1, e_1), (s_2, e_2) \in M$, it holds that $s_1 \leq s_2$ iff $e_1 \leq s_2$.

**(2)** For each variable $v$, child variable $v'$ of $v$, and index $i$, all intervals in $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathbf{sm}}(v)[i])$ are *grounded* in $\overset{D,Q}{\mathbf{sm}}$. An interval $(s, e) \in \mathsf{intervals}_{v'}(\overset{D,Q}{\mathbf{sm}}(v)[i])$ is grounded in $\overset{D,Q}{\mathbf{sm}}$ if $s \leq e \leq |\overset{D,Q}{\mathbf{sm}}(v')|$.

**(3)** For each $v \neq \mathsf{root}(Q)$, the sequence $\overset{D,Q}{\mathbf{sm}}(v)$ is consistent with the order associated with $\mathsf{rel}(v)^D$ in $D$.

**(4)** Finally, we record for each sequence map, the set of edges in $Q$ covered by that sequence map. This set of edges is denoted by $\mathsf{edgeCover}(\overset{D,Q}{\mathbf{sm}}) \subset (\mathrm{dom}\,\overset{D,Q}{\mathbf{sm}})^2 \cap \mathsf{Edges}(Q) \subset \mathsf{Vars}(Q)^2$. A sequence map may only contain references between bindings for variable pairs contained in its edge cover: For each pair of variables $v, v' \in \mathsf{Vars}(Q)$ with binding indices $i \leq |\overset{D,Q}{\mathbf{sm}}(v)|, i' \leq |\overset{D,Q}{\mathbf{sm}}(v')|$ such that $\overset{D,Q}{\mathbf{sm}}(v')[i'] \in \mathsf{Nodes}_{v'}(\overset{D,Q}{\mathbf{sm}}(v)[i])$, it must hold that $(v, v') \in \mathsf{edgeCover}(\overset{D,Q}{\mathbf{sm}})$.

Together the above restrictions guarantee that, with each binding of a variable $v$, there are at most associated $|N|$ intervals over the bindings for $v' \in \mathsf{children}(v)$ in a sequence map: The *set* of associated intervals does not contain duplicate intervals (since it is a set), each interval covers at least one index, no two intervals overlap, and all intervals are grounded. Thus we can, at most, have $|\overset{D,Q}{\mathbf{sm}}(v')| \leq |N|$ intervals each covering one of the bindings of $v'$ in the sequence map.

To illustrate the notion of sequence map consider again the query in Figure 53. The sequence map representing all answers to that query against the data in Figure 42 is illustrated in Figure 54: From bindings for "Emperors" (i.e., members of the unary type relation Emperor) we reference related bindings for name and province (where we use the name of the province instead of the ID for ease of presentation). The relations are expressed as intervals associated with the abbreviated query variable (N for name, P for province). Observe, that since the data is CIG shaped and the bindings are ordered accordingly we need always only single intervals. We abbreviate single element intervals as standard pointers.

It is worth emphasizing that we allow *multiple* intervals to represent the related bindings for a child variable. This is necessary to represent answers to queries on arbitrary graphs. As discussed in Sections 11.3.1 and 11.3.2, we can guarantee a single continuous interval for more restrictive shapes

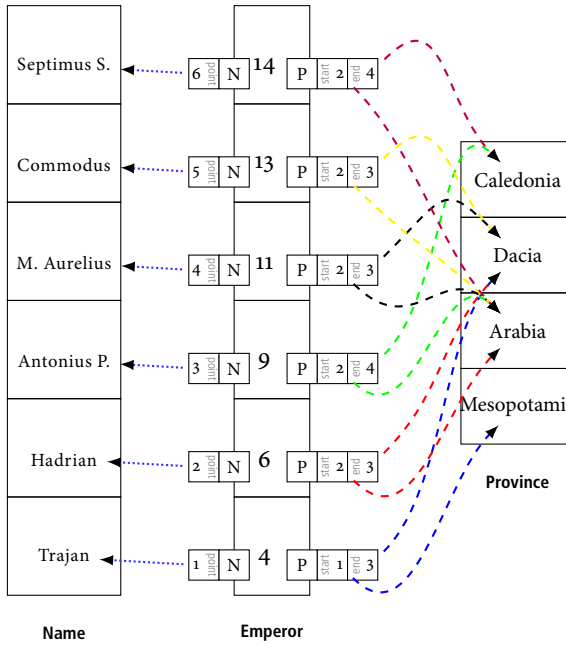



Figure 54. Sequence Map: Example. For the query from Figure 53 on the data of Figure 42.

of data, viz. trees, forest, and CIGs. Even for arbitrary graphs the use of interval pointers is beneficial in many cases, cf. Section 11.3.3.

A sequence map serves as a compact representation of an answer relation. This relation can be retrieved from a sequence map as follows:

**Definition 11.5** (Induced relation)**.** Let $\overset{D,Q}{\mathbf{sm}}$ be a sequence map with associated edge cover EC. Let $U = \operatorname{dom} \overset{D,Q}{\mathbf{sm}} = \{v_1, \ldots, v_k\} = \{v \in \mathsf{Vars}(Q) : \exists n \in \mathsf{Nodes}(D) : n \in \overset{D,Q}{\mathbf{sm}}(v)\}$. Then $\overset{D,Q}{\mathbf{sm}}$ induces a relation $\overset{D,Q}{R_{\mathbf{sm}}}$ such that

$$\overset{D,Q}{R_{\mathbf{sm}}} = \Big\{(b_1, \ldots, b_k) \in \mathsf{Nodes}(D)^k : \exists k \in \mathbb{N} : \overset{D,Q}{\mathbf{sm}}(v_i)[k] = b_i \wedge \\ ((v_i, v_j) \in \mathrm{EC} \implies b_j \in \mathsf{Nodes}_{v_j}(\overset{D,Q}{\mathbf{sm}}(v_i)[k]))\Big\}$$

Note, that this implies that for pairs of variables that are not in the edge cover of the sequence map *all* combinations of bindings are included in the induced relation.

In Section 12.7, we introduce the sequence map extraction that allows to extract bindings of one or more variables from a sequence map in form of a relation. If bindings for all variables in $S$ are extracted, this operation yields exactly the induced relation.

The empty relation is induced by the empty sequence map, denoted by $\overset{D,Q}{\mathbf{sm}}_{\varnothing}$. It is also induced by a sequence map where some of the "links" imposed by the query are missing, e.g., if for some child variable all bindings of its parent variable include no interval pointers to bindings of the child variable. In this sense, such a sequence map and the empty sequence map can be considered equivalent:

**Definition 11.6** (Equivalent sequence maps)**.** Let $\overset{D,Q}{\mathbf{sm}}_1$ and $\overset{D,Q}{\mathbf{sm}}_2$ be two sequence maps. We call $\overset{D,Q}{\mathbf{sm}}_1$ and $\overset{D,Q}{\mathbf{sm}}_2$ equivalent, if they induce the same relation. However, they may differ, e.g., in the chosen order of elements, the intervals used, etc.

Note, that a sequence map does not need to map all variables in $\mathsf{Vars}(Q)$. If some variables are not mapped, the resulting answer is incomplete w.r.t. any constraints in $Q$ involving the missing variables. In other words, absent variables are ignored when considering the induced relation of a sequence map. In the following section, we define a class of sequence maps, called complete sequence maps, that represents an answer relation to a full query rather than to only a part of a query.

### 11.2.1 CONSISTENT AND INCONSISTENT SEQUENCE MAPS

Sequence maps store (intermediary) answers to tree queries decomposed. They are an exponentially more succinct store than flat relations. On flat

relations, if we restrict bindings of one variable we implicitly affect bindings for all variables since each tuple in the answer relation represents one particular assignment of bindings to query variables. Assume, e.g., we drop all tuples where the binding of $v$ has a value $\leq 10$. A binding $b'$ for some other variable $v'$ may, however, only occur together with bindings for $v$ that have value $\leq 10$. Thus dropping the above tuples also drops all occurrences of $b'$.

*Explicit propagation and temporary inconsistency*

On sequence maps, however, bindings are stored per variable (or column of the flat relation). Thus, when we modify bindings for one variable, bindings for other variables are not implicitly affected, rather these changes must be *explicitly propagated*. Details of this propagation are discussed later in Section 12. Intuitively, in the above case $b'$ remains among the bindings of $v'$ still pointing (assuming, for simplicity, that $v'$ is the parent variable of $v$) to the now dropped bindings for $v$. However, $b'$ could be dropped without loosing any proper answer (there is no way to extend $b'$ bindings for $v'$ to full answers). In a sense, such a sequence map is *inconsistent* as there are bindings for $v'$ that refer to invalidated (or "bombed") bindings for $v$. We can address this in two ways: **(1)** *Immediate propagation:* All operations on a sequence map that restrict variable bindings for one variable (or column) ensure before the conclusion of the operation that those restrictions are propagated to all possibly affected variables. **(2)** *Separate propagation from (local) restriction:* Operations on a sequence map may restrict bindings of one or more variables without immediately propagating the effect to other connected variables. The advantage is that we can perform a series of restrictions on one or even a subset of the query variables and only at the end of that series propagate all changes at once. The disadvantage is that we have to mark temporary inconsistencies and must ensure that they are propagated at some point. However, we can simulate the first case by following each operation on the sequence map immediately by a propagation to all related variables.

*Inconsistent sequence map*

In the following, we adopt the second approach since it is more flexible and requires, for many queries, significantly less propagation operation. To support this approach we introduce the concept of inconsistent sequence map, i.e., sequence maps where some restrictions on variable bindings have not yet been fully propagated.

To distinguish invalidated variable bindings, we mark each invalidated variable binding with a failure marker $\lightning$ from a (finite) set of failure markers $\mathcal{B}$[3] with $|\mathcal{B}| \leq |\mathsf{Nodes}(D)|$. Furthermore, we extend the signature of a

3 For consistency, we use a set of failure markers to be able to continue to consider a sequence as duplicate free.

sequence map to include failure markers:

$$\overset{D,Q}{\mathbf{sm}} : \mathsf{Vars}(Q) \to \mathit{SubSeq}((\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q)\times \mathit{Intervals}}) \cup \mathcal{B})$$

We limit the number of failure markers by the number of nodes in $D$, since failed bindings only result from invalidating existing bindings and a sequence map can, for a single variable, contain at most $|\mathsf{Nodes}(D)|$ bindings. Note also, that we do not need to record related bindings (for child variables) if a binding is "bombed". Also, by definition of the induced relation of a sequence map, failure markers do not affect the induced relation as all bindings in a induced relation must be nodes from $D$.

Failure markers address the invalidation of entire bindings for a variable (e.g., due to additional unary constraints). Another form of local restriction, however, is the removal of references to bindings of a child variable in the bindings of a parent variable. E.g., the province of Mesopotamia in the sequence map of Figure 54 is related only to emperor Trajan. If we remove the reference from Trajan to Mesopotamia, i.e., change the $P$ interval of the emperor with ID 4 to $2-3$ instead of $1-3$. (or "bomb" Trajan), Mesopotamia is not any more part of any full answer to the query and could thus be dropped. So, again, the sequence map retains information that, if we propagate immediately, could be dropped. We call bindings such as Mesopotamia *dangling bindings*. A dangling binding $n$ is *direct*, if there is no binding for the immediate parent variable with an interval pointer covering $n$. Otherwise it is *indirect*.

**Definition 11.7** (Consistent and inconsistent sequence maps)**.** A sequence map $\overset{D,Q}{\mathbf{sm}}$ is called *inconsistent*, if **(1)** it contains any "failed" bindings $\lightning \in \mathcal{B}$, or **(2)** it contains any "dangling" binding, i.e., a binding $b \in \overset{D,Q}{\mathbf{sm}}(v)$ for some variable $v$ such that $v' = \mathsf{parent}(v)$, $v' \in \mathrm{dom}\,\overset{D,Q}{\mathbf{sm}}$, and there is no $i \in \mathbb{N}$ such that $b \in \mathsf{Nodes}_v(\overset{D,Q}{\mathbf{sm}}(v')[i])$. Otherwise it is called *consistent*.

The notion of equivalent sequence maps immediately extends to inconsistent sequence maps (note that the bindings in the induced relation are from $\mathsf{Nodes}(Q)$ excluding $\lightning$). In Section 12.5.3 we introduce an algorithm for efficiently propagating changes in a sequence map. Using this algorithm, we obtain the following result:

**Theorem 11.1.** *Let $\overset{D,Q}{\mathbf{sm}}$ be an inconsistent sequence map. Then there is a consistent sequence map $\overset{D,Q}{\mathbf{sm}}'$ equivalent to $\overset{D,Q}{\mathbf{sm}}$. This sequence map can be computed in $\mathcal{O}(\tilde{q} \cdot n \cdot i)$ where $\tilde{q} = |\mathrm{dom}\,\overset{D,Q}{\mathbf{sm}}|$, $n = |\mathit{Nodes}(D)|$, and $i$ the maximum number of intervals per binding in $\overset{D,Q}{\mathbf{sm}}$. For tree, forest, and CIG data $i = 1$.*

*Proof.* See Section 12.5.3. □

Figure 55 illustrates a more complex sequence map than the one from Figure 54. It is inconsistent since there are some failure markers and binding $d_{12}$ for variable $v_3$ is dangling. Furthermore, if the failure markers are propagated $d_3$ for $v_1$ is also dropped (there are no proper related bindings for $v_2$ which in consequence makes also $d_{11}$ in $v_3$ dangling.

Even if we consider only consistent sequence maps, there are multiple sequence maps with the same induced relation: They contain the same bindings for each variable, but the interval pointers between bindings may vary as long as they cover the same set of bindings. E.g., in one sequence map a binding may contain $\{(v, 1, 3)\}$ to point to bindings of $v$, in another $\{(v, 1, 1), (v, 2, 2), (v, 3, 3)\}$, and in yet another $\{(v, 1, 2), (v, 3, 3)\}$. However, there is a unique *minimal* interval representation for the interval pointers of each binding (here the one of the first sequence map):

**Definition 11.8** (Interval-minimal sequence map)**.** A sequence map $\overset{D,Q}{\mathsf{sm}}$ is called *interval-minimal*, if the set of interval pointers for any binding in $\overset{D,Q}{\mathsf{sm}}$ is minimal, i.e., there is no smaller set of interval pointers that covers the same bindings for each child variable.

**Theorem 11.2.** *A* interval-minimal, consistent *sequence map is uniquely identified by its induced relation, i.e., there is no other* interval-minimal, consistent *sequence map with the same induced relation.*

*Proof.* Let $R$ be the induced relation of a given sequence map $S$. Then, **(1)** we can not add or remove a binding for any variable $v \in \operatorname{dom} S$: If we add a binding for a single variable, the binding is dangling and the resulting sequence map is not consistent. If we add a binding and reference if from some interval pointer of some binding for the parent variable, the induced relation is no longer the same but contains additional tuples. **(2)** we can not extend or shrink, collapse or divide an interval pointer for a binding $n$ of $v \in \operatorname{dom} S$. If we extend an interval pointer $i$ and the added indices (which are adjacent to $i$) are covered in the old interval set, then the original set of intervals is not minimal (as we could collapse $i$ with the interval pointer covering the adjacent indices). If the added indices are not covered in the old interval set, we introduce new tuples into the induced relation. Analog for shrinking, we remove tuples in the induced relation. For collapsing and dividing, either the original interval set is not minimal (if we can collapse) or the resulting interval set is not minimal (if we divide). □

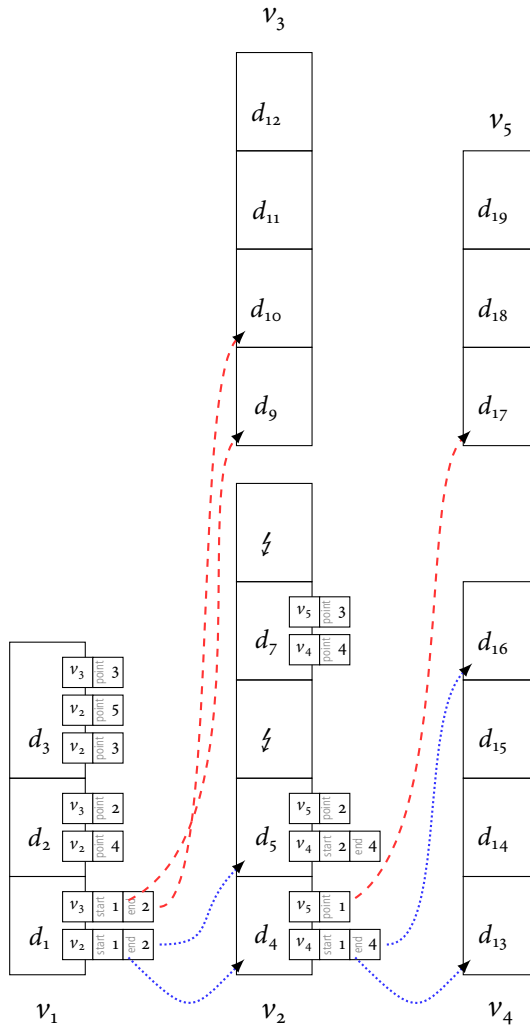



Figure 55. Inconsistent Sequence Map.

### 11.2.2 ANSWERS: CONSISTENT AND COMPLETE SEQUENCE MAPS

Sequence maps may be only a partial mapping of query variables to bindings and thus contain only partial or intermediary answers to a query, many of which may actually not contribute to any complete answer of the query.

**Definition 11.9** (Complete sequence map)**.** For a tree query $Q$ a sequence map $\overset{D,Q}{\mathsf{sm}}$ is called *complete* if **(1)** all variables of $Q$ are covered by the sequence map: $\mathsf{dom}\,\overset{D,Q}{\mathsf{sm}} = \mathsf{Vars}(Q)$; **(2)** all relations of $Q$ are covered in all bindings for the involved variables: for all $v, v' \in \mathsf{Vars}(Q)$ it holds that $v = \mathsf{parent}(v')$ if and only if, for all $i \leq |\overset{D,Q}{\mathsf{sm}}(v)|$, $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathsf{sm}}(v)[i]) \neq \varnothing$; **(3)** all relations in the sequence map are covered by the query: for all $v, v' \in \mathsf{Vars}(Q)$ it holds that if there is a $i \in \mathbb{N}$ such that $\mathsf{intervals}_{v'}(\overset{D,Q}{\mathsf{sm}}(v)[i]) \neq \varnothing$, then $v = \mathsf{parent}(v')$.

Complete sequence maps can still be inconsistent, though failure markers can only occur for leaf variables (for inner variables, they violate condition (2) of the definition as there are no related bindings to a failure marker by definition). "Dangling" bindings may occur for any variable.

In Chapters 13, we show how to translate a tree query $Q$ into sequence map operations in such a way, that the resulting sequence map represent the correct answers to $Q$ if evaluated against $D$. The resulting sequence maps are always both consistent and complete.

However, before we can compile queries into sequence map operations, we first need to define those operations in Chapter 12. Before we can define the operations, we establish a number of properties for sequence maps on different shapes of data in Section 11.3. With these properties, we can establish, in Section 11.4 bounds for the space needed to represent (intermediary) results of a tree query in a sequence map for the different kinds of data. These results are used, finally, to define operations on sequence maps such as creation of sequence maps, projection, join, union, subtraction, and propagation to remove inconsistencies, cf. Section 12. We conclude with a number of variants of the sequence map, in particular, a purely relational sequence map.

## 11.3 ON THE INFLUENCE OF DATA SHAPE

Sequence maps are capable of representing (intermediary) answers to tree queries on any relational structure as defined above. However, when we pose certain restrictions on the shape of the relations involved, we can place bounds on the number of intervals required to represent relations

between bindings of adjacent query nodes and thus on the size of the sequence map.

### 11.3.1 EXPLOITING TREE-SHAPE OF DATA: SINGLE INTERVAL POINTERS

Tree data exhibits a number of regularities that have been exploited by most previous approaches to querying structural relations on tree data: labeling schemes such as pre-/post-encoding [86, 116]allow constant time, constant space membership test of tree-shaped relations as well as their closure (descendant or ancestor in XPath); similarly, twig join approaches [48] exploit the fact that in tree shaped data, their is at most one parent for each node in the data and at most $d$ related nodes, with $d$ depth of the tree, if the closure relation is considered, i.e., at most $d$ ancestor nodes; like twig joins, we exploit limits on the number of parents and ancestors of a node in SPEX. Additionally, we observe that, for horizontal closure relations, the bounds are less favorable, e.g., breadth of tree for following-sibling and size of tree for following. But also for horizontal relations we observe that the images for nodes follow certain patterns (e.g., the followings of a node are a subset of the followings of all its pre-order predecessors).

In the following, we give alternative characterisations of tree-shaped relations and their transitive closures that allow us to identify in what way we can order the domain to allow a linear representation of the related nodes in the given relation. Furthermore, we use these characterisations to show in Section 11.3.2 how to go beyond tree data and still maintain the linear representation.

*Direct structural relations in trees: image disjointness*

To start with, let us consider relations that directly form trees, i.e., direct structural relations such as child, next-sibling, and next (in the sense of [112]). If we look at how the images of nodes are shaped under these relations, we can note that the images of two different nodes never overlap. We call this property *image disjointness*:

**Definition 11.10** (Image Disjointness Property)**.** Let $R$ be a binary relation over some domain $N$. Then $R$ is said to carry the IMAGE DISJOINTNESS property, if and only if, for any two nodes $n_1 \neq n_2 \in N$, it holds that $R(n_1) \cap R(n_2) = \varnothing$.[4]

We choose this formulation of image disjointness as it immediately induces a set of orders on $N$ that guarantee that the image of a node $n \in N$ can be represented as a single interval: keep each $R(n)$ together (with

4 Recall, the we denote with $R(n) = \{n' \in N : (n, n') \in R'\}$.

arbitrary "internal" order) but choose an arbitrary order among the $R(n)$. More formally, Let $<_{arbitrary}$ be some arbitrary total order on $N$, $<_{\text{dom}}$ some total order on $\text{dom}\, R$, and $<_n$ a total order on each $R(n)$ for $n \in \text{dom}\, R$. Then $\prec$ is a total order on $N$ with

$$\prec = \left\{(n, n') \in (\text{rng}\, R)^2 : n \in R(m) \wedge n' \in R(m') \wedge \begin{cases} n <_m n' & \text{if } m = m' \\ m <_{\text{dom}} m' & \text{if } m \neq m' \end{cases}\right\}$$

All $n \notin \text{rng}\, R$ either follow or precede the $n \in \text{rng}\, R$.

It is easy to see that a sequence over $N$ consistent with any such order allows the representation of the images of any node $n$ under $R$ as a single interval.

In terms of orders on the tree, any breadth-first traversal induces such an order. Whether the traversal is top-down or bottom-up, left-to-right, or right-to-left is immaterial. In fact, the order in which the nodes of the tree are visited is entirely arbitrary as long as the children of each node are visited together.

The image disjointness property captures exactly all tree- and forest-shaped relations:

**Theorem 11.3.** *Let $R$ be a binary relation over some domain $N$. Then $(N, R)$ is a forest iff $R$ carries the image disjointness property.*

*Proof.* Recall, that a (directed rooted) tree is a rooted connected simple graph with a unique simple path between the root and any other node. A forest is a disjoint union of trees, i.e., we all multiple roots but each node is part of exactly one tree. In other words, there is only one root per node from which that node is reachable and, as in a tree, there is a unique simple path between that root and the node.

If $(N, R)$ is a forest and $n_1 \neq n_2 \in N$, then $R(n_1) \cap R(n_2) = \varnothing$: any node $n' \in R(n_1) \cap R(n_2)$ violates the forest property as it is either, if $n_1$'s root is different from $n_2$, reachable from multiple roots (i.e., $(N, R)$ is no disjoint union of trees) or, if $n_1$ and $n_2$ have the same root, there are two unique simple paths between the root and $n'$ by appending $n'$ to the path from the root to $n_1$ and to $n_2$.

If $R$ has the image disjointness property, then let $\mathit{roots}(R) = \{n \in N : \nexists\, n' \in N : (n', n) \in R\}$. **(1)** Each node $n \in N$ is reachable from at most one root $r \in \mathit{roots}(R)$: If $n$ is directly reachable (a child) from $r$ then it can not be directly reachable from any other $n' \in N$ including any $r \in \mathit{roots}(R)$ due to the image interval property and as $r$ is not reachable from any node in $n$ by definition of $\mathit{roots}(R)$. If $n$ is indirectly reachable from $r$ the same argument can be made recursively for the nodes in the path from $r$ to $n$. **(2)** For each node $n \in N$, there is a unique path from its root $r$ to $n$. If

there are two distinct paths from $r$ to $n$ then either $n$ or some node on the path from $r$ to $n$ lies in the image of two distinct nodes in violation of the image interval property. □

*Structural closure relations in trees: image containment*

However, on tree data also closure relations can be represented and tested in linear time and space, e.g., the descendants of a node. In fact, for any of the structural *closure* relations such as descendant, ancestor, following, following-sibling, etc. we can observe the same. However, the images of two distinct nodes are clearly not distinct in these cases. E.g., the descendants of a child are always a subset of the descendants of its parent. For all these relations, the images of two different nodes can overlap, but on tree data only in a "disciplined" manner: either they do not overlap at all or one is entirely contained in the other. We formalize this property as follows:

**Definition 11.11** (Image Containment Property)**.** Let $R$ be a binary relation over some domain $N$. Then $R$ is said to carry the IMAGE CONTAINMENT property, if and only if, for any two nodes $n_1 \neq n_2 \in N$, it holds that $R(n_1) \cap R(n_2) \neq \varnothing$ implies that $R(n_1) \subset R(n_2) \vee R(n_2) \subset R(n_1)$.

Again, given this property we can readily define an order over $N$ such that the images of each node can be represented as a single, continuous interval: the containment of images constitute a hierarchy on the domain of $R$. As long as that hierarchy is respected by the order, the images of a node can be represented as a single, continuous interval on a sequence over $N$ consistent with that order.

More formally, let $<_n$ be some order on $R(n)$ for each $n \in \operatorname{dom} R$. Let $\mathit{direct}(n)$ for some $n \in \operatorname{rng} R$ be the $n' \in \operatorname{dom} R$ such that $n \in R(n')$ and there is no $n'' \in \operatorname{dom} R$ such that $n \in R(n'')$ and $R(n'') \subset R(n')$. Finally, let $<_{\mathrm{dom}}$ be an arbitrary order on $\operatorname{dom} R$ and $<_{\mathit{incl}} = \{(n, n') \in (\operatorname{dom} R)^2 : R(n) \subset R(n') \vee (R(n) \cap R(n') = \varnothing \wedge n <_{\mathrm{dom}} n')\}$. Then $\prec$ is a total order on $N$ with

$$\prec = \{(n, n') \in (\operatorname{rng} R)^2 : m = \mathit{direct}(n) \wedge m' = \mathit{direct}(n') \wedge \begin{cases} n <_m n' & \text{if } m = m' \\ m <_{\mathit{incl}} m' & \text{if } m \neq m' \end{cases}$$

All $n \notin \operatorname{rng} R$ either follow or precede the $n \in \operatorname{rng} R$.

The image containment property captures exactly closure relations over forest-shaped base relations:

**Theorem 11.4.** *Let $R$ be a binary relation over $N$. Then $R$ carries the image containment property if and only if there is a forest-shaped base relation $R'$ such that $R$ is the transitive closure of $R'$.*

*Proof.* If $R$ is the transitive closure of a forest-shaped relation $R'$ and $m \in R(n), m \in R(n')$. Then either $n$ is an ancestor of $n'$ or the other way round as $R'$ is a forest, i.e., $R(n) \subset R(n')$ or vice versa.

If $R$ carries the image containment property, let $R' = \{(n, n') \in N^2 : R(n') \neq \varnothing \wedge R(n') \subset R(n) \wedge \nexists\, m \in N : R(n') \subset R(m) \wedge R(m) \subset R(n)\} \cup \{(n, n') \in N^2 : n' \in R(n) \wedge \nexists\, m \in N : n' \in R(m) \wedge R(m) \subset R(n)\}$. The first set represents the hierarchy of the inner nodes, the second set the leaf nodes. Then $(N, R')$ is a forest, since **(1)** leaf nodes $l$ have by definition a single parent: if there are two $n \neq n' \in N$ with $l \in R(n)$ and $l \in R(n')$ but neither $R(n) \subset R(n')$ nor $R(n') \subset R(n)$ (otherwise $n$ or $n'$ do not fulfill the condition for parents of lead nodes), then $R(n) \cap R(n') \neq \varnothing$ but neither is subset of the other in violation of the image containment property of $R$. **(2)** inner nodes $i$ have also a single parent: if there are two nodes $n, n'$ with $i \subset R(n)$ and $i \subset R(n')$ but no $m, m' \in N$ with $i \subset R(m) \subset R(n)$ and $i \subset R(m') \subset R(n')$, then $R(n) \cap R(n') \neq \varnothing$, yet neither is a subset of the other. Thus, the image containment property is violated. □

### 11.3.2 BEYOND TREES: CONSECUTIVE ONES PROPERTY

Tree data, as argued above, allows us to represent relations on that data more compact, e.g., using various interval-based labeling schemes. Here, we introduce a new class of graphs, called *continuous-image graphs* (or CIGS for short), that generalize features of tree data in such a way that we can evaluate (tree) queries on CIGS with the same time and space complexity as techniques such as twig joins [48] which are limited to tree data only. Moreover, we show that even skipping and pruning techniques used for tree data carry over to continuous-image graphs (cf. Section 12.9.1).

*Characterization of continuous image graph*

Recall from the previous chapter that continuous-image graphs are a proper superset of (ordered) trees. On trees we require that each node has at most one parent. For continuous-image graphs, however, we only ask that we can find a single order on all nodes of the graph such that the children of each parent form a continuous interval in that order.

*CIGS: Image interval property*

Formally, we define a CIG (in Section 10.2) as an arbitrary relation (or graph) that carries the following property:

**Definition 11.12** (Image interval property)**.** Let $R$ be a binary relation over some domain $N$. Then $R$ is said to carry the IMAGE INTERVAL property, if and only if there exists a total order $<_i$ on $N$ with its induced sequence $S$ over $N$ such that for all nodes $n \in N$, $R(n) = \varnothing$ or $R(n) = \{S[s], \ldots, S[e] : s \leq e \in \mathbb{N}\}$.

This property merely formalizes the above observation that one means of compacting a binary relation on $N$ (for linear space storage with linear

time membership test) is through the use of intervals. Here, we demand that there is an order and thus a sequence over $N$ that allows us such an interval representation.

The image interval property is a generalization of both image disjointness and image containment but characterizes a strictly larger class of relations.

**Theorem 11.5.** *Let $R$ be a binary relation over some domain $N$. If $R$ carries the image disjointness (image containment) property, then $R$ also carries the image interval property.*

*Proof.* For relation with image disjointness or image containment property, we choose an order over $N$ as described when defining the two properties. Together with any such order, $R$ fulfils the definition of image interval property. □

**Theorem 11.6.** *The image interval property covers a strictly larger class of relations than either the image disjointness or the image containment property.*

*Proof.* Figure 42 as well as Figure 56 show relations that carry the image interval property but do not carry either image disjointness or image containment. In general, CIGs allow more freedom in overlapping than relations with image containment property: two nodes may share some but not all nodes in their image. However, as discussed above, cf. Figure 41, they still pose a limit on the sharing. □

*Deciding the image interval property*

It is easy to see, that the weakest of the three properties, image disjointness, can be tested in quadratic time over the size of the domain. Surprisingly, the same holds for the image interval property: For that we observe that it is merely another formulation of to the consecutive-ones property introduced for $\{0,1\}$ matrices in [97]. A $\{0,1\}$ matrix is said to exhibit the *consecutive-ones* property if its rows may be permuted in such a way as to make the ones in each column consecutive.

**Theorem 11.7.** *Let $R$ be a binary relation over some domain $N$. Then $R$ carries the image interval property iff its adjacency matrix carries the* consecutive-ones *property.*

*Proof.* Recall, that the adjacency matrix of a binary relation $R \subset N^2$ is a quadratic $\{0,1\}$ matrix $M$ where rows and columns correspond to the nodes in $N$ and $M(i,j) = 1$ iff the nodes corresponding to the $i$th row and $j$th column stand in relation $R$. Let $M_R$ be a corresponding matrix for a relation $R$. $M_R$ exhibits the consecutive-ones property if and only if the image interval property holds for $R$: Each column represents the images of

a node. A permutation of the rows is merely a specific order of the nodes in $N$. Thus, if a permutation of the rows exists such that the ones in each column are consecutive, then an order on the nodes in $N$ exists such that the images of each node form a single continuous interval on the sequence of nodes represented by the row permutation. □

For the consecutive-ones problem [40] gives the first linear time (in the size of the matrix) algorithm based on so called PQ-trees, a compact representation for permutations of rows (or columns) in a matrix. Consider the relation represented in Figure 56 as an adjacency matrix. It is neither forest-shaped (e.g., 1 has many parents: $(A, 1)$, $(A, 2)$, ...) nor does it carry the image containment property (2 is in image of both $A$ and $B$ but neither image is a subset of the other).

However, we can compute a PQ-tree for this relation that represents all permutations of column orders such that the 1s are consecutive in each row. This PQ-tree is shown in Figure 57. A PQ-tree contains, as the name suggests, two kinds of inner nodes: Q nodes and P nodes. P nodes indicate that any permutation of its children guarantees consecutive 1s in each row. Q nodes indicate that its children must be traversed in order or in the inverse order. For Figure 57 this means, that the PQ-tree represents the permutations 4271356, 4271536, 4217356, 4217536, 6357124, 6351724, 6537124, 6531724.

Figure (b) shows one such permutation where, indeed, the 1s are consecutive in all rows. It is also easy to see that we can flip 7 and 1, as well as 3 and 5 arbitrarily (they are each identical). And, of course, we can invert the order of the columns without violating the consecutive ones property.

The PQ-tree algorithm gives us a quadratic decision algorithm whether any given relation carries the image interval property (details cf. [40]).

**Theorem 11.8.** *Let $R$ be a binary relation over some domain $N$. Then deciding whether $R$ carries the image interval property and computing a corresponding order $<_i$ has space and time complexity $\mathcal{O}(|N|^2)$*

More recent refinements of the PQ-tree, viz. the PC-tree described in [129, 131], give a slightly simpler test for the consecutive-ones property (albeit with the same complexity). Figure 58 shows the PC-tree for the relation in Figure 56. In a PC-tree, we have again two kinds of inner nodes: P nodes where we can freely permute the children and C (or cyclic) nodes where we can only traverse the children either in clockwise or in counterclockwise order.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| B | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| C | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

(a) Adjacency matrix representation of a sample relation

| | 4 | 2 | 7 | 1 | 3 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| B | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| C | 0 | 0 | 1 | 1 | 1 | 1 | 1 |

(b) Consecutive-ones permutation of relation from (a)

Figure 56. Relation with (1-) interval property

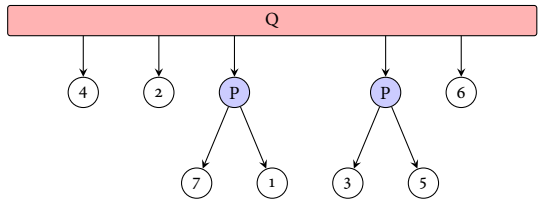



Figure 57. PQ-Tree for Relation in Figure 56

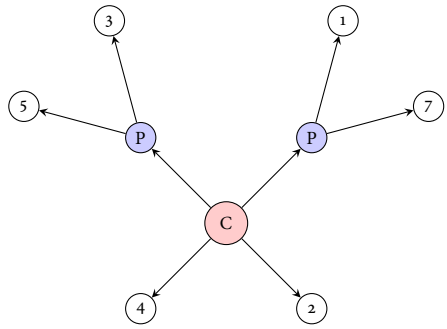



Figure 58. PC-Tree for Relation in Figure 56

### 11.3.3 OPEN QUESTIONS: BEYOND SINGLE INTERVALS

With the image interval property, we now have a characterisation of a large class of graphs that contains all forest-shaped relations and their closures, yet covers a substantially larger class of graphs. This characterisation allows us, as for forest-shaped relations, to use a single interval to represent the related nodes of any node in a relation carrying the property. Moreover, deciding whether a relation carries that property is decidable in quadratic time and gives, as a by-product, an appropriate order. Note, that the decision is entirely independent of the query and can thus be computed when storing the relation rather than an query evaluation time.

*Bounded Intervals: k-Interval Property*

The image interval property ensures that we need at most one interval to represent the image of a node under a relation. Linear time representation and linear time membership test, however, can also be achieved if we relax that a bit more: If there is a (preferably small) $k \in N$ independent of $|N|$ that represents an upper limit to the number of intervals needed to represent the image of a node, that still gives us a linear space representation and linear time membership test.

Thus, even for a relation $R$ that carries the $k$-interval property for a small $k \in N$ we can still profit from the sequence map representation over a plain relational representation:

**Definition 11.13** (Image $k$-interval property)**.** Let $R$ be a binary relation over some domain $N$ and $k \in \mathbb{N}$. Then $R$ is said to carry the $k$-IMAGE INTERVAL property, if and only if there exists a total order $<_i$ on $N$ with its induced sequence $S$ over $N$ such that for all nodes $n \in N$, $R(n) = \varnothing$ or

$$R(n) = \{S[s_1], \ldots, S[e_1], S[s_2], \ldots, S[e_2], \ldots, S[s_l], \ldots S[e_l] : s_i \leq e_i \wedge l \leq k\}.$$

If we modify the relation as in Figure 59 by adding 4 to the image of $A$ and 6 to the image of $C$, the resulting relation does no longer carry the consecutive-ones property (and thus has no associated PQ-tree). It carries, however, the 2-interval property as illustrated by the permutation in Figure 59. Furthermore, it also has a corresponding PC-tree (in fact, the same as the original relation). This is due to the fact that PC-trees actually test for the *circular-ones* property *circular-ones* property, as defined in [131]. The circular ones property of a $\{0, 1\}$-matrix indicates that there is a permutation of the rows such that either the 0s or the 1s are consecutive in each column. It is called "circular ones" as, the 1s are still consecutive in all columns if we consider the matrix circular, i.e., after the last row we continue with the first row.

Whether there is a polynomial decision algorithm for the general $k$-interval property, remains an open question though the test for "circular ones" points towards a decision algorithm for $k = 2$. In the following, we use only the 1-interval property characterising CIG data.

*Arbitrary, but Optimal Intervals for Arbitrary Graphs*

Even on arbitrary graphs, where we can no longer guarantee liner space representation and linear time membership test, we nevertheless can often profit from selecting a suitable order on the nodes of $N$: for most graph the number of intervals needed to represent the images of a node is often much lower than the worst case of $\mathcal{O}(|N|)$.

To characterise such an order, we first define the (minimal) interval representation for a relation $R$:

**Definition 11.14** (Minimal interval representation of a relation)**.** Let $R \subset N^2$ be a (finite) binary relation and $<$ a total order over $N$. Let $S_<$ be the induced sequence over $N$ for $<$. Then $I : \operatorname{dom} R \to 2^{Intervals}\}$ is called the *minimal interval representation* of $R$ over $<$ if $(s, e) \in I(n)$ for any $n \in domR$ iff for all $s < i < e$: $(n, S[i]) \in R$ (all elements in the interval are in $R(n)$), $s = 1$ or $(n, S[s-1]) \notin R$ and $e = |S_<|$ or $(n, S[e+1]) \notin R$ (there is no larger interval over $S_<$ that also fulfils the first condition and includes $(s, e)$).

Note, that the intervals in $I(n)$ are non-overlapping and cover $R(n)$ for each $n \in \operatorname{dom} R$. They are non-overlapping, as $(s_1, e_1), (s_2, e_2) \in I(n)$ with $s_1 < s_2 < e_1$ either $e_2 \leq e_1$ which means $(s_2, e_2)$ violates the second condition of the definition or $e_2 > e_1$ which means both violate the second condition and $(s_1, e_2)$ must be included in $I(n)$ by definition.

Using the notion of interval representation, we can now define the minimal interval representation and thus the cost of representing a relation by intervals over some given order $<$.

**Definition 11.15** (Interval cost of a relation)**.** Let $I \in \mathcal{I}_<(R)$ be the minimal interval representation of a relation $R$ under $<$. Then we call $\mathbf{IN}_<(R) = \sum_{n \in N} |I(n)|$ the size of $I$ and the *interval cost* for $R$ under $<$.

This gives us the cost of representing a relation $R$ under an order on $N$. What we would like to find, is an order on $N$ with the lowest cost for the representation:

**Definition 11.16** (Interval-optimal order)**.** Let $R \subset N^2$ be a (finite) binary relation and $\mathcal{O}$ the set of all total orders over $N$. Then an order $< \in \mathcal{O}$ is called *interval optimal* if its associated interval cost $\mathbf{IN}_<(R)$ is minimal among the interval costs of all orders over $R$.

Obviously, we can find the interval-optimal order by trying each permutation of the nodes in $N$. As for the $k$-interval property, it is an open question whether there is a *polynomial* decision algorithm for the optimal interval order.

For a given order $<$, however, we can compute a minimal interval representation in polynomial time by Algorithm 1.

**Algorithm 1**: Compute interval representation from relation

**input** : Relation $R$ total order $<$ over rng $R$.
**output**: Interval representation $I$ of $R$.

```
1  S ← induced sequence for < over rng R;
2  I ← ∅;
3  foreach n ∈ dom R do
4      Intervals ← ∅;
5      start ← ⊥;
6      for i ← 1 to |S| do
7          if (n, S[i]) ∈ R and start = ⊥ then
8              start ← i ;
9          if (n, S[i]) ∉ R and start ≠ ⊥ then
10             Intervals ← Intervals ∪ {(start, i − 1)} ;
11             start ← ⊥ ;
12     if start ≠ ⊥ then
13         Intervals ← Intervals ∪ {(start, |S|)} ;
14     I ← I ∪ {(n, Intervals)};
15 return I
```

**Theorem 11.9.** *Algorithm 1 computes the minimal interval representation of the input relation $R$ under the order $<$ in time $\mathcal{O}(n^2)$ where $n = |Nodes(R)|$.*

*Proof.* Let $I$ be the result of Algorithm 1 for a given $R$ under the order $<$ and $S_<$ the induced sequence over rng $R$ for $<$. Then $I$ is the minimal interval representation of $R$ under $<$ as **(1)** there is an interval $(s, e) \in I(n)$ for each $n' \in R(n)$ such that $s \leq S^{-1}(n') \leq e$ (as at each $n' \in R(n)$ either a new interval is started, l. 7–8, or an open interval is continued); **(2)** for all intervals $(s, e) \in I(n)$ and $s < i < e$, $S_<[i] \in R(n)$ (as if $S_<[i] \notin R(n)$ the previous interval is closed at $i - 1$, l. 9–11, or start $= \bot$); **(3)** for all intervals $(s, e) \in I(n)$ either $s = 1$ or $S_<[s-1] \notin R(n)$ (if $S_<[s-1] \in R(n)$ then $s$ can not be the start of an interval as the interval either starts at $s - 1$, l. 7–9, or before); **(4)** for all intervals $(s, e) \in I(n)$ either $e = |S_<|$ or $S_<[e+1] \notin R(n)$

(only if $S_<[e+1] \notin R(n)$ or we are at the end of the sequence is an interval added with the current interval position as end index).

Algorithm 1 runs in $|\operatorname{dom} R| \cdot |\operatorname{rng} R| = \mathcal{O}(|\mathsf{Nodes}(R)|)$: the induced sequence can be computed in $\mathcal{O}(n \log n)$. The main loop (l. 3–14) iterates over all elements in $\operatorname{dom} R$, the inner loop over the elements of $S$ which are all elements in $\operatorname{rng} R$. □

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| B | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| C | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

(a) Adjacency matrix representation of a sample relation with 2-interval property

| | 4 | 2 | 7 | 1 | 3 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| B | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| C | 1 | 0 | 1 | 1 | 1 | 1 | 1 |

(b) Permutation of relation from (a) illustrating the circular-ones property

Figure 59. Relation with 2-interval property

## 11.4 SPACE BOUNDS FOR SEQUENCE MAPS

With the above properties of data represented in sequence maps, we can now give precise characterisations of the space used by a sequence map $\overset{D,Q}{\mathsf{sm}}$ over a relational structure $D$ and a query $Q$.

**Theorem 11.10.** *Let $\overset{D,Q}{\mathsf{sm}}$ be a sequence map over a relational structure $D$ and a query $Q$. Then the size of $\overset{D,Q}{\mathsf{sm}}$ is bounded by $\mathcal{O}(q \cdot n \cdot i) \leq \mathcal{O}(q \cdot n^2)$ where $n = |\mathit{Nodes}(D)|, q = |\mathit{Vars}(Q)|$, and $i$ is the maximum number of intervals needed to represent the image of a node $n \in N$ under any query relation $R$ and the order $O(R)$ associated with $R$ in $D$.*

For trees and CIGs, this result immediately gives linear space complexity for the sequence map as shown in the following Section 11.4.1.

*Proof.* Recall, the signature of a sequence map from Section 11.2:

$$\overset{D,Q}{\mathsf{sm}} : \mathsf{Vars}(Q) \to \mathit{SubSeq}(\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q) \times \mathit{Intervals}})$$

Each of the $q$ variables in $\mathsf{Vars}(Q)$ is mapped to a sub-sequence over $\mathsf{Nodes}(D)$ each of which is associated with a subset of $\mathsf{Vars}(Q) \times \mathit{Intervals}$. Since sequences are by definition (see Section 11.2) duplicate-free, the size of each sub-sequence over $\mathsf{Nodes}(D)$ is bounded by $n = |\mathsf{Nodes}(D)|$. Thus, we have at most $q \cdot n$ bindings represented in a sequence map. However, each binding is associated with a sub-set over $\mathsf{Vars}(Q) \times \mathit{Intervals}$ the size of which is bounded by $q \cdot i$ where $i$ is the maximum number of intervals needed to represent the image of a node $n \in N$ under any query relation $R$ and its associated order $O(R)$.

This indicates a bound of $q \cdot n \cdot q \cdot i$. However, for each variable $v$, its bindings are referenced only from bindings of its parent variable $v'$, but not from the bindings of any other variable. Thus, for each variable, we have at most $n \cdot i$ intervals referencing bindings of that variable and the total size of the associated interval sets is limited by $q \cdot n \cdot i$.

In total, we arrive at a bound of $q \cdot n$ for the bindings and a bound of $q \cdot n \cdot i$ for the interval sets and thus at an overall bound of $q \cdot n + q \cdot n \cdot i = \mathcal{O}(q \cdot n \cdot i)$.

Note, that $i \leq n$, as all intervals are, by definition of a sequence map, non-overlapping and grounded: Since they are grounded, the largest index covered by any interval is bounded by $n$ (the maximum length of the sequence of child variable bindings). Since they are non-overlapping, there are at most $n$ intervals to cover a sequence from 1 to $n$, viz. $n$ intervals of size 1.

The edge cover associated with a sequence map does not affect the space complexity: since $Q$ is a tree query, there are at most $q - 1$ edges in $Q$ and thus in the edge cover of a sequence map for $Q$. □

Note, that the above result holds for arbitrary relations and tree queries. It shows that the sequence map provides a polynomial storage for (intermediary) answers of tree queries on arbitrary relations, i.e., an exponentially more succinct storage than flat relations.

### 11.4.1 LINEAR SPACE BOUNDS FOR TREES AND CIGS

Together with the results from the previous sections we can immediately infer a number of conditions when the sequence map provides linear data complexity for answer storage:

**Corollary 11.1.** *Let* $\overset{D,Q}{\mathbf{sm}}$ *be a sequence map over a relational structure* $D$ *and a query* $Q$*. Let all query relations carry the image disjointness, image containment, or image interval property together with their associated order* $O(R)$ *in* $D$*. Then the size of* $\overset{D,Q}{\mathbf{sm}}$ *is bounded by* $\mathcal{O}(q{\cdot}n)$ *where* $n = |\mathit{Nodes}(D)|$ *and* $q = |\mathit{Vars}(Q)|$*.*

*Proof.* For relations that fulfill one of these properties, $i = 1$ since the $R(n)$ can be represented as a single interval for each $n \in \mathsf{Nodes}(D)$ over the induced sequence for $O(R)$. □

Since image disjointness, containment and interval are precise characterisations of tree/forest-shaped relations, closure relations over tree/forest-shaped relations, or CIG-shaped relations resp., we can also state this result as follows:

**Corollary 11.2.** *Let* $\overset{D,Q}{\mathbf{sm}}$ *be a sequence map over a relational structure* $D$ *and a query* $Q$*. Let all query relations be tree- or forest-shaped, closure relations over tree- or forest-shaped relations, or* CIG*-shaped. Then the size of* $\overset{D,Q}{\mathbf{sm}}$ *is bounded by* $\mathcal{O}(q \cdot n)$ *where* $n = |\mathit{Nodes}(D)|$ *and* $q = |\mathit{Vars}(Q)|$*.*

Note, if all variables in $Q$ are answer variables (in other words, if we evaluate $Q$ by pattern matching in the classification of [190]), this bound becomes tight if we ensure that only nodes from $N$ are retained in the sequence map that are actual matches for the full query, cf. Section 12.9 for details and conditions on the data needed to ensure that property.

Analogously, for relations with $k$-interval property we have at most $k$ intervals needed to represent the image of each node an thus $\mathcal{O}(q \cdot n \cdot k)$ space bound.

**Corollary 11.3.** *Let* $\overset{D,Q}{\mathbf{sm}}$ *be a sequence map over a relational structure* $D$ *and a query* $Q$*. Let all query relations carry the* $k$*-image interval property together with their associated order* $O(R)$ *in* $D$*. Then the size of* $\overset{D,Q}{\mathbf{sm}}$ *is bounded by* $\mathcal{O}(q \cdot n \cdot k)$ *where* $n = |\mathit{Nodes}(D)|$ *and* $q = |\mathit{Vars}(Q)|$*.*

Finally, note that these results hold also for inconsistent sequence maps as the number of failure markers is bounded by $|N|$ as well as the number of "dangling" bindings (since the latter are nodes from $N$).

The space bounds for sequence maps established in this section form the foundation for the time complexity of the sequence map operations discussed in Section 12. Before, we turn to the sequence map operations and thus the ClQCAG algebra proper, we finish our discussion of the sequence map data structure by a brief outlook on variants of the sequence map: a purely relational variant of the sequence map and a variant of the sequence map supporting not only tree queries but also some graph queries, viz. diamond-free DAG queries.

## 11.5 SEQUENCE MAP VARIATIONS

### 11.5.1 PURELY RELATIONAL SEQUENCE MAP

As defined above, the sequence map uses sequences to represent bindings and associates with each binding a set of variable-interval pairs. Both are features that are not provided by pure relational databases (though sequences and order, e.g., are provided in SQL and most practical DBS).

**Definition 11.17** (Purely relational sequence map)**.** A purely relational sequence map, denoted $\overset{D,Q}{\mathbf{sm}}*$, over a relational structure $D$ and a query $Q$ is a set of relations: For each variable $v$ in $Q$, there is a relation $R_v \in \overset{D,Q}{\mathbf{sm}}*$ with schema $\langle \mathit{index} \in \mathbb{N}, \mathit{binding} \in \mathsf{Nodes}(D), \mathit{child\ variable} \in \mathsf{children}(v), \mathit{start\ index} \in \mathbb{N}, \mathit{end\ index} \in \mathbb{N} \rangle$. Each tuple in $R_v$ stores a node binding together with its index and one interval of related nodes for one child variable.

In contrast to the definition of a sequence map in Section 11.2, we duplicate the index and binding node information for each related child variable interval. To obtain the related bindings of some child variable $v'$ for a given binding of the parent variable $v$, we evaluate a *range query* on $R_{V'}$ for all tuples with *index* between *start index* and *end index*.

Though general range queries require $\mathcal{O}(n \log n)$ time to iterate all nodes in a given interval, we can exploit in this case the fact that the *index* column can be stored as a sequence or ordered relation allowing indexed access and $\mathcal{O}(n)$ time iteration. In this sense, the sequence map as defined in Section 10.3 can be seen as a specific realisation of the purely relational sequence map with linear time iteration for related bindings of a given node.

### 11.5.2 MULTI-ORDER SEQUENCE MAP FOR DIAMOND-FREE G QUERIES

The above definitions of a sequence map allow only tree queries (or tree cores of arbitrary queries). However, we can in fact extend the sequence map approach also to forest queries and even certain classes of DAG queries. In fact, the above definitions are also amenable to forest queries. Beyond forest queries, we can still use the sequence map for *diamond-free DAG queries*, albeit with a slight modification. Diamond-free DAG queries are queries in the shape of a DAG where there are no two distinct paths between two nodes (and thus no diamond-shaped sub-graph). Diamond-free DAG queries are also used, e.g., in [171, 169] (there named single-join DAG queries).

If we consider diamond-free DAG queries, there may be nodes in the query that have multiple parents. However, as for tree queries only the parent and child nodes are relevant to decide whether a data item is a match for a query node. The multiple parents, however, may be connected using different relations, e.g., XPath's child, descendant, and following relations. In the above definitions, we only demand that for each relation the continuous-image property of the data holds. If we have DAG queries this might lead to different, possibly incompatible orders for the image relations (e.g., child needs a breadth-first order to obtain continuous-images vs. depth-first order for descendant). However, we do not need to "strengthen" the CIG property for diamond-free DAG queries. Rather, we use the purely relational variant of the sequence map but add in $R_v$ one separate index column for each different incoming relation of $v$ with incompatible orders and interval pointers are resolved as range queries over the appropriate order number.

The downside of this adaptation is that we can now no longer use the index for ordered storage of the relation as there are several index columns. Therefore, we have to fall back to general, $\mathcal{O}(n \log n)$ time range queries rather than indexed access. For most of the operations discussed in Section 12, this increases the time complexity with a logarithmic factor. Moreover, range queries are in most practical SQL database systems not very efficient, cf. [119], if compared with indexed access.

# 12 SEQUENCE MAP OPERATORS



## 12.1 INTRODUCTION AND OVERVIEW

In this chapter, we concentrate on the first set of operators in the CIQCAG algebra, the operators for constructing and manipulating a sequence map. The remaining operators are discussed in Chapter 13. The sequence map operators, summarized in Table 58, roughly fall into three groups: initialization or access operators (Section 12.3) create a sequence map from a given relation, combination operators (Section 12.4) such as join, union, and difference combine two sequence maps, and reduction operators (Section 12.5) such as projection, selection, or propagation drop some of the bindings contained in a sequence map. The role of the "odd men out" is played by the extraction operator (Section 12.7) that returns parts of the

| access | join (conjunction) | union | difference |
|---|---|---|---|
| $\ddot{\mu}_{v,v'}(D,Q)$, $\ddot{\mu}_{v}(D,Q,R)$ | $\ddot{\bowtie}_{\cap}^{(\lightning)}$, $\ddot{\bowtie}^{(\lightning)}$, $\ddot{\ltimes}^{(\lightning)}(S,S')$ | $\ddot{\cup}(S,S')$ | $\ddot{\setminus}(S,S')$ |
| **projection, rename** | **selection** | **propagation** | **expansion** |
| $\ddot{\pi}_{V}(S)$, $\ddot{\rho}_{v_1 \to v_2}(S)$ | $\ddot{\sigma}_{c}^{(\lightning)}(S)$ | $\ddot{\omega}_{v}^{\uparrow}(S)$, $\ddot{\omega}_{v}^{\downarrow}(S)$ | $\mathcal{F}_{V}(S)$ |

Table 58. Overview of sequence map operators in CIQC AG (all operators return a single sequence map $S$ except $\mathcal{F}$ which returns a (standard) relation)

induced relation of its input sequence map and plays the bridge between sequence maps, used in the evaluation of the tree core of a query, and relations, used in the evaluation of the remaining query, if there is any.

The sequence map operators closely mirror their relational counterparts, where such exists. In general, they are defined by reduction to the relational counterpart on the induced relation(s) of the input sequence map(s). In contrast to relations, however, sequence maps are, in general, not closed under union, difference, or even projection, see Section 12.4. This is addressed by placing certain restrictions on the input sequence maps for each of these operators.

The three most unusual operators are the sequence map join, propagation, and extraction. For the two latter operators, the reason is mostly that there are no relational counterparts. For the sequence map join the main reason is that we present a number of variants for the sequence map join with different characteristics. Most notably, we introduce (as we also do for selection and initialization) both consistent and inconsistent variants (marked with a $\lightning$ as superscript) of the join: The former ensures that a resulting sequence map is consistent if the input sequence maps are, the latter allows and in many cases introduces inconsistencies. However, as discussed in Section 12.4.1, the use of the inconsistent variant actually yields, in general, an evaluation plan with lower complexity. This is the case, as we propagate inconsistencies once per query variable rather than at each join.

*Query example*

To illustrate, how these operators play together to implement a typical Web query consider again the query in Figure 52 and the spanning tree for that query shown in Figure 60. Note, that we mark also $v_2$ as answer node (i.e., with a red rectangle ▭). This is necessary as both $v_2$ and $v_4$ are used in the non-tree part of the query. We can evaluate this (spanning) tree query using sequence maps with many different sequence map expressions (see

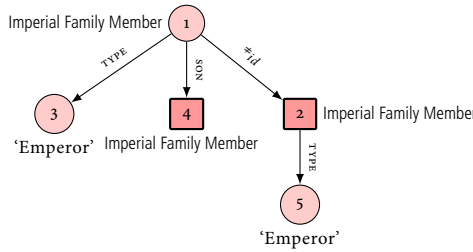



Figure 60. Spanning tree for the query from Figure 52

Section 12.8 for equivalences between ClQCAG expressions formed from sequence map operators).

One approach is to use the, in most cases more efficient, inconsistent variants of the operators where possible, at the price of additional propagation operations at the end of the query:

$$F_{v_2,v_4}\Big(\ddot{\omega}^{v_2}_{\downarrow}(\ddot{\omega}_{v_4}^{\uparrow}( (\ddot{\mu}^{\not\downarrow}_{v_1,v_4}(D,Q) \ddot{\ltimes}^{\not\downarrow} \ddot{\pi}_{v_1}(\ddot{\mu}^{\not\downarrow}_{v_1,v_3}(D,Q) \ddot{\ltimes}^{\not\downarrow} \ddot{\mu}_{v_3}(D,Q,\text{'Emperor'}))) \ddot{\bowtie}^{\not\downarrow} \ddot{\mu}^{\not\downarrow}_{v_1,v_2}(D,Q) \ddot{\ltimes}^{\not\downarrow} \ddot{\pi}_{v_2}(\ddot{\mu}^{\not\downarrow}_{v_2,v_5}(D,Q) \ddot{\ltimes}^{\not\downarrow} \ddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\ )$$

Instead of the semi-join operators we could also choose (inconsistent) join operators. However, we could not replace the inconsistent joins with consistent ones without adapting also the contained expressions as consistent join operators require that the input sequence maps are consistent.

A variant using only consistent operators is, nevertheless, also possible and, as expected, even more compact:

$$F_{v_2,v_4}\Big( (\ddot{\mu}_{v_1,v_4}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\pi}_{v_1}(\ddot{\mu}_{v_1,v_3}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\mu}_{v_3}(D,Q,\text{'Emperor'}))) \ddot{\bowtie}_{\cap} \ddot{\mu}_{v_1,v_2}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\pi}_{v_2}(\ddot{\mu}_{v_2,v_5}(D,Q) \ddot{\bowtie}_{\cap} \ddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\ \Big)$$

Here, we use a join variant (denoted by a $\cap$ subscript) that allows only input sequence maps with disjoint edge covers and is slightly more efficient than the general join, $\ddot{\bowtie}$, which could be employed here as well.

For more examples, see Section 12.8 and 12.9.

*Sequence-at-a-time vs. iterator scheme*

The sequence map operators are introduced as sequence-at-a-time operators, i.e., given one or more input sequence maps they compute all their results at once. In Section 12.9, we show how to obtain a iterator-based evaluation where results are generated tuple-at-a-time. The disadvantage of that scheme is that we are no longer free to rearrange the expression

of the CIQCAG expression, but rather cluster all expressions modifying values for a certain variable. Also, for general graph or even CIG queries the benefit of a tuple-at-a-time evaluation is fairly low as there are no bounds on the number of nodes related to a single node under a given relation and thus all previously computed answers may still be related to nodes (matching with their parent variable) that come "later" in the processing.

On tree or forest data, however, there are such bounds: Namely, the number of parents of a node is limited by 1 for direct tree relations, i.e., relations with image disjointness property, and by $d$ for closure relations, i.e., relations with image containment property, where $d$ is the depth of the base relation. However, the presence of closure relations still prevents effective pruning of already considered bindings in a sequence map unless we assume that the orders associated with the incoming relations of all pairs of parent-child variables are consistent, i.e., if a node $n_1$ is before another node $n_2$ in the parent order, all the nodes in $n_1$'s image are before or the same as the nodes in $n_2$'s image. An example of such relations are child and descendant from XPath and, actually, all pairs of base relation and corresponding closure relation. For queries containing only such order-compatible relations, we can prune already considered bindings earlier than in the general case and arrive at a tighter bound for the space used by their evaluation, viz. $\mathcal{O}(q \cdot d)$ which coincides with the lower bound for such queries shown in [190].

## 12.2 INTERVAL ACCESS TO A RELATIONAL STRUCTURE

Before we turn to the actual operators, we briefly outline the physical storage for a relational structure:

In Section 11.2, we define a relational structure $D$ as a tuple $(R_1^D, \ldots, R_k^D, O^D)$ over some relational schema $(R_1[U_1], \ldots, R_k[U_k])$. $O^D$ associates with each binary relation $R_i^D$ a total order on the (finite) domain.

For the description of the CIQCAG operators, we assume a *specific way of accessing* a given relational structure (which provides the logical description of the queried data):

**(1)** For each order $<_N \in \operatorname{rng} O^D$, we access the nodes of $D$ in the induced sequence of $<_N$ over $\mathsf{Nodes}(D)$. Recall, that $<_N$ is a total order and thus has a unique induced sequence that faithfully represents that order. Intuitively, instead of querying pairs of nodes $(n_1, n_2)$ with $n_1 <_N n_2$ we iterate over all nodes of $D$ such that a node $n$ is accessed at index $i_n$ where $n_1 <_N n_2$ iff $i_{n_1} < i_{n_2}$.

For each induced sequence of an order $<_R = O^D(R)$ we also store the "flip" index $\mathsf{flip}_{<_R} \in \mathbb{N}$ such that, for all $i \leq \mathsf{flip}_{<_R}$, $S_{<_R}[i] \in \operatorname{rng} R$ and, for

all $j > \mathsf{flip}_{<_R}, S_{<_R}[j] \notin \operatorname{rng} R$. Recall, from the definition of a relational structure in Section 11.2, that in the associated order $<_R$ of each relation $R$ all nodes in $\operatorname{rng} R$ precede all nodes not in $\operatorname{rng} R$. Thus, each induced sequence has a (unique) flip index.

**(2)** Each binary relation $R$ is accessed through its minimal interval representation $I(R)$ over its associated order $O^D(R)$. Recall, that a minimal interval representation maps each node in $n \in \operatorname{dom} R$ to the minimal set of intervals needed to represent $R(n)$ over the induced sequence of $O^D(R)$.

From Theorem 11.9 and the fact that the induced sequence of a total order can be computed in $\mathcal{O}(n \log n)$, we conclude that, even if we store the relations and orders in $D$ as sets of pairs (or tables), we can provide the above access in polynomial time:

**Corollary 12.1.** *Let $D = (R_1^D, \ldots, R_k^D, O)$ be a relational structure and $m$ be the maximum cost of membership test for $R_1^D, \ldots, R_k^D$. Then its physical storage can be computed in $\mathcal{O}(k \cdot |\mathit{Nodes}(D)|^2 \cdot m)$.*

If all relations in $D$ are extensional, we assume $m$ to be constant and thus $\mathcal{O}(k \cdot |\mathsf{Nodes}(D)|^2)$ as bound for the computation of the physical storage.

In the following, we assume that the iteration over the induced sequences is in $\mathcal{O}(n)$, membership test is constant, and the iteration over the interval representations $I(n)$ of a single node is in $\mathcal{O}(|I(n)|)$.

These time bounds can be achieved for arbitrary relations with associated order $<$ by storing the induced sequence for each order (at the same cost as storing the orders itself) and storing for each binary relation its minimal interval representation. The minimal interval representation $I_<$ can be stored in $\mathcal{O}(|I_<|) \leq \mathcal{O}(n \cdot \max\{|I(n)| : n \in \operatorname{dom} R\} \cdot \log n)$ space (if we assume constant pointer size, we can drop the logarithmic factor).

For tree, forest, and CIG data, this yields linear space in the number of nodes in $D$, for arbitrary relations the same or less space than storing the relation as a table of pairs.

It should also be noted, that the above time and space bounds can be achieved for structural relations over tree or forest data using an interval labeling of the elements in the tree, e.g., pre/post-encoding [86, 116] or BIRD [200]: Given two nodes, we can determine in constant time whether they are child, parent, descendant, ancestor, following, preceding, following-sibling, preceding-sibling, etc. just from looking at the node labels. We can also iterate over all, e.g., descendants of a node by a single range query on the pre- and post-values. The advantage of such labeling schemes is that they encode an entire set of structural relations using only $n \cdot l$ where $l$ is the size of a label space. For most of these encodings, $l$ is in the same order as the size of a pointer to a sequence over $n$, i.e., $l \leq \log n$.

In the following, we record bindings for all query variables in the order associated with the relation the *incoming* edge of the variable is labeled with in the query. For root variables we use some fixed, but arbitrary order $\prec_{std}$.[1] Thus, in a query $Q$ against a relational structure $D$, we can associate with each query variable $v \in \mathsf{Vars}(Q)$ an order $\prec_v$ such that

$$\prec_v = \begin{cases} \prec_{std} & \text{if } v \text{ root variable} \\ O^D(\mathsf{rel}(v_1)^D) & \text{otherwise} \end{cases}$$

### 12.2.1 STORING AND MANAGING INTERVAL SETS

Bindings in a sequence map may be associated with a set of intervals and their associated child variables, i.e., a relation over $\mathsf{Vars}(Q) \times \mathbb{N} \times \mathbb{N}$. For each binding and child variable, the intervals are non-overlapping. We assume that this set is stored partitioned by child variable and, for each such variable, the non-overlapping intervals are in order of their *start index*.

For the sequence map operations, we introduce four algorithms on such interval sets (or, where convenient, on interval sets for a single child variable): Adapt, Algorithm 8, slides and/or shrinks a set of intervals with the help of a change set, that maps indices of the sequence those intervals reference to a indices of a subsequence of that sentence such that the same binding is at the position of the index in the original sequence as at the position of its image in the subsequence. For details, see Section 12.4.1. JoinInts, Algorithm 9, computes the minimal interval sets representing the join (or intersection) of the two interval sets. DifferenceInts, Algorithm 14, computes the minimal interval set representing the difference of two interval sets. UnionInts, Algorithm 25, computes the minimal interval set representing the union of two interval sets. All three algorithms construct the new interval set ordered by increasing start index.

Note, that all these algorithms are linear in the size of the input sets as they can exploit **(1)** that the interval sets are stored in order of their *start index* and **(2)** that the intervals in each set are non-overlapping. The ordered storage allows us to avoid sorting the intervals. Thus, the complexity for these algorithm does not contradict results on interval merge or union, e.g., in [128].

1 The fixed order ensures that over several constraints for the same root variable bindings are recorded in the same order. This is exploited in the (merge) join algorithm discussed in Section 12.4.1.

## 12.3 INITIALIZE (FROM RELATION)

The basic operation when constructing a sequence map representing the answers of a tree query is the *initialization* of a sequence map from a single given relation such that the resulting sequence map represents the given relation. There are two variants of the initialization operation, one for unary relations and one for binary relations. These are separate as for binary relations we need to store not only bindings for the two query variables involved but also interval pointers between those relations.

*Initialization of unary relations*

Initializing a sequence map with a unary relation is essentially the same as turning a unary relation into a sequence over the order associated with its query variable. The result is a *consistent* sequence map such that its induced relation represents the given unary relation:

**Definition 12.1** (Initialization of unary relations)**.** Let $D$ be a relational structure, $Q$ a tree query, $v \in \mathsf{Vars}(Q)$, and $R$ a relation with relation name in $\mathcal{L}_V^D(v)$. Then, $\dddot{\mu}_v(D, Q, R)$ returns a *consistent* sequence map $\overset{Q,D}{\mathbf{sm}}$ such that

**(1)** $R$ is the induced relation of $\overset{Q,D}{\mathbf{sm}}$ (represents the unary relation on $v$) and

**(2)** $\overset{Q,D}{\mathbf{sm}}|_v = \overset{Q,D}{\mathbf{sm}}$ (contains bindings only for $v$).

The edge cover associated with $\dddot{\mu}_v(D, Q, R)$ is empty.

Algorithm 2 computes such a sequence map for a given input relation. The resulting sequence map is consistent as there are no failure markers (the algorithm does not introduce any and the nodes in $S$ are distinct from any failure marker) and no dangling bindings (as there are bindings in the sequence map only for $v$ and, thus, the parent of $v$, if there is any, is not in the sequence map's domain).

*Dropping consistency*

At first glance, asking for a consistent sequence map is the obvious choice for this operator. However, e.g., when the selectivity for a unary relation is very low, we can actually profit from an inconsistent sequence map containing an entry for all nodes in $D$ but failure markers where those entries are not in the given unary relation as this slightly simplifies, e.g., the join algorithm (cf. Section 12.4.1). As with all inconsistent versions of the ClQcAG algebra, the price is that to obtain correct answers, we need to add specific propagation operators (at the end of the processing), see Section 12.5.3.

We denote with $\dddot{\mu}_v^{\lightning}(D, Q, R)$ the sequence map initialization where we allow also inconsistent sequence maps and realize this operation by replacing lines 1–7 as follows:

$S \leftarrow$ induced sequence for $\prec_v$ over $\mathsf{Nodes}(D)$ ;

**Algorithm 2**: $\ddot{\mu}_v(D, Q, R)$

**input** : Relational structure $D$, tree query $Q$, variable $v \in \mathsf{Vars}(Q)$, *unary* relation $R$ with relation name in $\mathcal{L}_V^D(v)$.

**output**: Consistent sequence map representation of $R$ and associated edge cover.

```
1 S ← induced sequence for ≺_v over Nodes(D) ;
2 S' ← ∅ ;
3 j ← 1 ;
4 for i ← 1 to |S| do
5     if S[i] ∈ R then
6         S'[j] ← S[i];
7         j ← j + 1 ;
8 return {(v, S')}, ∅
```

```
foreach i ← 1 to |S| do
    if S[i] ∉ R then
        S[i] ← ↯
return {(v, S)}
```

Here, we employ *in-place* editing of the sequence $S$ and merely "bomb" entries not in the given relation with a failure marker.

**Theorem 12.1.** *Both, $\ddot{\mu}_v(D, Q, R)$ and $\ddot{\mu}_v^{\lightning}(D, Q, R)$, can be computed in $\mathcal{O}(n \cdot m)$ where $n = |\mathsf{Nodes}(D)|$ and $m$ is the cost for the membership test in $R$.*

*Proof.* Algorithm 2 computes $\ddot{\mu}_v(D, Q, R)$. It loops over $|S| = |\mathsf{Nodes}(D)|$ elements of $S$. For each element, the membership in $R$ is tested. The same reasoning applies to the modified version of Algorithm 2 for computing $\ddot{\mu}_v^{\lightning}(D, Q, R)$. □

*Initialization of binary relations*

For a binary relation, we not only need to record bindings for two variables, the relations parent and child variable, but also the interval pointers referencing related bindings from bindings of the parent variable to bindings of the child variable. Again, we require that the resulting sequence map is consistent.

**Definition 12.2** (Initialization of binary relations)**.** Let $D$ be a relational structure, $Q$ a tree query, and $v_1, v_2 \in \mathsf{Vars}(Q)$. Then $\ddot{\mu}_{v_1,v_2}(D, Q)$ returns a *consistent* sequence map $\overset{Q,D}{\mathsf{sm}}$ such that

**(1)** $\mathsf{rel}(v_2)^D$ is the induced relation of $\overset{Q,D}{\mathsf{sm}}$ (represents the relation between $v_1$ and $v_2$) and

**(2)** $\overset{Q,D}{\mathsf{sm}}|_{v_1,v_2} = \overset{Q,D}{\mathsf{sm}}$ (contains bindings only for $v_1$ and $v_2$).
The associated edge cover for $\dddot{\mu}_{v_1,v_2}(D,Q)$ is $\{(v_1,v_2)\}$.

Recall, that by the definition of a sequence map (part (3)), the bindings of $v_1$ in $\dddot{\mu}_{v_1,v_2}(D,Q)$ are ordered by $O(\mathsf{rel}(v_1)^D)$ and those of $v_2$ by $O(\mathsf{rel}(v_2)^D)$.

Algorithm 3 computes such a sequence map in linear time:

**Algorithm 3**: $\dddot{\mu}_{v_1,v_2}(D,Q)$

**input** : Relational structure $D$, tree query $Q$, variables $v_1, v_2 \in \mathsf{Vars}(Q)$.
**output**: Consistent sequence map representation of $\mathsf{rel}(v_2)^D$.

```
1  I  ← minimal interval representation of rel(v_2)^D ;
2  S_1 ← induced sequence for ≺_{v_1} over Nodes(D) ;
3  S'_1 ← ∅ ;
4  j ← 1 ;
5  for i ← 1 to |S_1| do
6      if I(S_1[i]) ≠ ∅ then
7          S'_1[j] ← (S_1[i], I(S_1[i])) ;
8          j ← j + 1 ;
9  S_2 ← induced sequence for ≺_{v_2} over Nodes(D) ;
10 return {(v_1, S'_1), (v_2, S_2|_{1,...,flip_{≺_2}})}, {(v_1, v_2)}
```

**Theorem 12.2.** *$\dddot{\mu}_{v_1,v_2}(D,Q)$ can be computed in $\mathcal{O}(n \cdot m_I)$ where $n = |\mathit{Nodes}(D)|$ and $m_I$ is the cost for accessing $I(n)$ for any node $n \in \mathit{Nodes}(D)$.*

*Proof.* Algorithm 3 computes a consistent sequence map with the induced relation $\mathsf{rel}(v_2)^D$: The sequence map is consistent as there are no failure markers (the algorithm does not introduce failure markers and the entries in $S_1$ and $S_2$ are nodes from $D$, not failure markers) and no dangling bindings (the parent variable of $v_1$, if there is any, is not mapped; for $v_2$, each binding has a corresponding binding of $v_1$ by definition of $\mathsf{flip}_{\prec_2}$). The induced relation of the sequence map is $\mathsf{rel}(v_2)^D$: For each pair $(n_1,n_2) \in \mathsf{rel}(v_2)^D$, $n_1$ is in $\overset{D,Q}{\mathsf{sm}}(v_1)$ by the construction of $S'_1$ in line 3–7; $n_2$ is in $\overset{D,Q}{\mathsf{sm}}(v_2)$ as all $n_2 \in \mathrm{rng}\,\mathsf{rel}(v_2)^D$ are associated with an index $\leq \mathsf{flip}_{\prec_2}$ by definition. Finally, since $I$ is an interval representation of $\mathsf{rel}(v_2)^D$ there is an interval in $I(n_1)$ and thus in $\overset{D,Q}{\mathsf{sm}}(v_1)$ such that the index associated with $n_2$ in $\overset{D,Q}{\mathsf{sm}}(v_2)$ is in that interval.

Given an interval representation of $\mathsf{rel}(v_2)^D$ and $m_I$ time access to $I(n)$ for each $n \in \mathsf{Nodes}(D)$, Algorithm 3 runs in time $\mathcal{O}(n \cdot m_I)$: it loops over

at most $n$ elements of $S_1$, for each obtaining $I(n)$. The restriction of $S_2$ to elements before $\mathsf{flip}_{<_2}$ can be done in $\mathcal{O}(|S_2|) \leq \mathcal{O}(n)$ time. □

*Dropping consistency*

As for the unary initialization, we can define a variant of the binary initialization that drops the consistency requirement from the above definition. We denoted this variant as $\ddot{\mu}^{\lightning}_{v_1,v_2}(D, Q)$. Algorithm 4 marks bindings for $v_1$ with no related bindings for $v_2$ with a failure marker (instead of dropping them) and does not limit $S_2$ to only those elements in $\mathrm{rng}\,\mathsf{rel}(v_2)^D$. It has obviously the same time complexity as the one computing a consistent sequence map, but operates *in-place*, i.e., without building a second sequence for bindings of $v_1$.

**Algorithm 4**: $\ddot{\mu}^{\lightning}_{v_1,v_2}(D, Q)$

**input** : Relational structure $D$, tree query $Q$, variables $v_1, v_2 \in \mathsf{Vars}(Q)$.
**output**: Sequence map representation of $\mathsf{rel}(v_2)^D$.

```
1 I ← minimal interval representation of rel(v2)^D ;
2 S1 ← induced sequence for ≺v1 over Nodes(D) ;
3 for i ← 1 to |S1| do
4     if I(S1[i]) ≠ ∅ then
5         S1[i] ← (S1[i], I(S1[i])) ;
6     else
7         S1[i] ← ↯ ;
8 S2 ← induced sequence for ≺v2 over Nodes(D) ;
9 return {(v1, S1'), (v2, S2)}, {(v1, v2)}
```

## 12.4 COMBINE

Once a sequence map is initialized by a single unary or binary relation, we need to be able to combine multiple sequence maps to evaluate a tree query containing more than one relation.

The essential operation for combining sequence maps is the *join* in analogy to a natural join on relations: take two sequence maps and, for all shared variables, retain bindings contained in both sequence maps, for variables occurring only in one sequence map, retain the bindings in those sequence map. In fact, this corresponds to a join of the induced relations of the two sequence maps and yields an important property of sequence maps: They are *closed under join*, i.e., the join of the induced relations of two sequence maps can be represented as a sequence map, cf. Theorem 12.9.

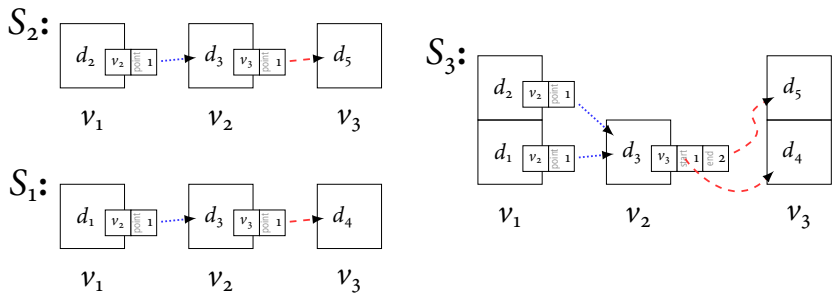



Figure 61. Sequence maps for illustrating "separate union" problem

There is no direct analog to the relational cross product, since a sequence map, in contrast to a relation, may not contain more than one sequence of bindings for a variable/attribute. If we join two sequence maps with disjoint domains, the result is, as in the relational case, the cross product (of the induced relations).

For the evaluation of proper tree queries, join is the only combine operation needed. However, we also investigate two more combine operations, *union* and *difference* with similar semantics to those for relations. These allow also queries where some parts are negated or alternatives to other parts to be evaluated in a sequence map (rather than on the level of flat relations, cf. Chapter 13). Intersection is omitted as it can be expressed through union and difference and, moreover, as the natural join yields intersection if the domain and edge covers of the input sequence maps are the same.

As for the corresponding operations on relations, we place certain restrictions on the shape of the sequence maps allowed in these operations: At least the involved sequence maps must map the same variables (i.e., the same nodes of the underlying query are covered) and edge cover (i.e., the same edges of the underlying query are covered). However, it turns out that this restriction does not suffice for combining sequence maps with union or difference:

Consider the sequence maps $S_1$ and $S_2$ in Figure 61 ($S_1 = \{v_1 \mapsto \{(1, d_1 \mapsto \{(v_2, 1, 1)\})\}, v_2 \mapsto \{(1, d_3 \mapsto \{(v_3, 1, 1)\})\}, v_3 \mapsto \{(1, d_4 \mapsto \varnothing)\}\}$, $S_2 = \{v_1 \mapsto \{(1, d_2 \mapsto \{(v_2, 1, 1)\})\}, v_2 \mapsto \{(1, d_3 \mapsto \{(v_3, 1, 1)\})\}, v_3 \mapsto \{(1, d_5 \mapsto \varnothing)\}\}$). The induced relations are $R_1 = \{(d_1, d_3, d_4)\}$ and $R_2 = \{(d_2, d_3, d_5)\}$ and $R_1 \cup R_2 = \{(d_1, d_3, d_4), (d_2, d_3, d_5)\}$. This is **(1)** different from the induced relation $\{(d_1, d_3, d_4), (d_1, d_3, d_5), (d_2, d_3, d_4), (d_2, d_3, d_5)\}$ of sequence map $S_3$ which is the result if we "union" the bindings for adjacent

pairs of variables independently (as we can do for the join). Nevertheless, $S_3$ is the smallest sequence map whose induced relation contains $R_1 \cup R_2$. **(2)** a relation that *can not be represented by any sequence map* as it does not exhibit a lossless-join decomposition into binary relations over each pair of adjacent variables. In fact, there are *no* multivalue dependencies in $R_1 \cup R_2$. Similar observations apply for difference, thus yielding the following result:

**Theorem 12.3.** *The union (difference) of the induced relations of two sequence maps is, in general, not an induced relation of any sequence map.*

To obtain a union and difference operation on sequence maps that is well-defined and intuitive w.r.t. the induced relations of the involved sequence maps, we restrict union and difference to single-variable or single-edge sequence maps: A single-variable sequence map contains bindings for a single query variable only (and, thus, has an empty edge cover). A single-edge sequence map contains bindings for two adjacent query variables $v$ and $v'$ and has an associated edge cover $\{(v, v')\}$.

On single-variable (single-edge) sequence maps, we can define union and difference operations such that the result is a single-variable (single-edge) sequence map and such that the union or difference of the induced relations of two single-variable (single-edge) sequence maps is the induced relation of the resulting sequence map.

### 12.4.1 JOIN

The first combination operator for sequence maps, $\vec{\bowtie}^{\not}_{\cap}(\overset{D,Q}{\mathbf{sm}}_1, \overset{D,Q}{\mathbf{sm}}_2)$, *joins* two sequence maps into one, such that the resulting sequence map represents the natural join of the two induced relations of the two input sequence maps. We first introduce a more restrictive variant of the general join that limits the overlapping between the edges covered by two sequence map. This allows us to carry over the interval pointers from the input sequence maps unchanged or only slightly adapted.

**Definition 12.3** (Sequence map join (disjoint edge covers))**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two sequence maps for $D$ over $Q$ such that *their associated edge covers are disjoint*. Then $\vec{\bowtie}^{\not}_{\cap}(S_1, S_2)$ returns a sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that

**(1)** the induced relation of $\overset{D,Q}{\mathbf{sm}}_3$ is the natural join of the induced relations of $S_1$ and $S_2$, i.e., $R_{\overset{D,Q}{\mathbf{sm}}_3} = R_{S_1} \bowtie R_{S_2}$.

**(2)** $\overset{D,Q}{\mathbf{sm}}_3|_{\operatorname{dom} S_1 \cup \operatorname{dom} S_2} = \overset{D,Q}{\mathbf{sm}}_3$ (contains bindings only for variables mapped either in $S_1$ or in $S_2$).

The associated edge cover for $\ddot{\bowtie}^{\not{\iota}}_{\cap}(S_1, S_2)$ is the *union* of the edge covers associated with $S_1$ and $S_2$.

Note that this definition yields a sequence map that leaves bindings for *non-shared* variables unchanged from either sequence map (correspond to attributes of their induced relations that occur only in one of the two relations, for these attributes the relational join subsumes to a cross product and thus retains any combination of the bindings). For *shared* variables, only those bindings are retained that occur in both sequence maps. This also applies to the (interval pointer) references from bindings of a parent variable $v$ to a child variable $v'$ of $v$: They are contained only in one of the sequence maps (due to the edge cover restriction), for the other sequence map the induced relation records *any* combination of bindings by definition (cf. Section 11.2).

*Edge covers: disjoint*

The restriction on the edge covers on $S_1$ and $S_2$ is imposed to ensure that for any pair of variables $v, v'$ only one of the sequence maps may contain interval pointers from $v$ to $v'$, though sequence maps may contain *bindings* for $v$ and $v'$. In other words, each *edge* of the query is enforced by at most one of the two sequence maps.

For a given CIQCAG expression, the edge cover of each sequence map (created as result of any sub-expression) can be determined statically, without knowledge about the data the expression is to be evaluated against: For each CIQCAG operation, we define here either how the edge cover is computed from its input (for combine and reduce operations) or the edge cover is independent from the input data (initialization). Thus, we can also statically determine whether a CIQCAG join expression is valid or violates the edge cover restriction defined above.

*Computing sequence map join with disjoint edge cover*

The above definition does not demand that the resulting sequence map is consistent. Therefore, Algorithm 5 computes a sequence map that represents the join of the induced relations as demanded in the definition of $\ddot{\bowtie}^{\not{\iota}}_{\cap}()$, but may be inconsistent: It "bombs" bindings not contained in both sequence maps rather than dropping them entirely. This has the effect that interval points can remain unchanged (but no reference an interval containing possibly bombed entries). Note, that interval pointers to bindings of a variable occur only in one of the two sequence maps as the incoming edge of each variable is unique (since the query is tree-shaped) and the edge covers are disjoint (and thus the unique incoming edge is only covered by one of the two sequence maps). This allows line 16 where we simply throw together intervals from both sequence maps. Finally, observe that by the definition of the initialization of a sequence map, bindings for the same query variable occur in the *same* order in all sequence maps for that query. Thus the bindings of a variable shared between the two sequence

**Algorithm 5:** $\overset{\cdots}{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$

**input** : Sequence maps $S_1$ and $S_2$ with *disjoint* edge covers
**output**: Sequence map res representing the join of the induced relations of the input maps

```
1  EC1 ← edgeCover(S1); EC2 ← edgeCover(S2);
2  AllVars ← dom S1 ∪ dom S2;
3  SharedVars ← dom S1 ∩ dom S2;
4  res ← ∅ ;
5  foreach v ∈ AllVars do
6      if v ∉ dom S2 then res ← res ∪ {(v, S1(v))} ;
7      else if v ∉ dom S1 then res ← res ∪ {(v, S2(v))};
8      else                                              // v is in both
           // 1 is the primary (fallback if v is in neither edge cover)
9          iter ← S1(v); alt ← S2(v) ;
10         if (v', v) ∈ EC2 for some v' then
               // v is sink in EC2, thus the order and number of entries
                  must be as in 2 (it can not be sink in EC1 as Q tree query
                  and edge covers disjoint)
11             iter ← S2(v); alt ← S1(v);
12         S ← ∅; i, j, k ← 1 ;
13         while i ≤ |iter| do
14             (n1, i) ← nextBinding(S1(v), i);
               (n2, j) ← nextBinding(S2(v), j); if n1 = n2 then   // Retain
               binding if same
15                 S[k] = (n1, intervals(iter[i]) ∪ intervals(alt[j])) ;
16                 i++; j++; k++;
17             else if n1 < n2 then          // "bomb" if in iter but not in alt
18                 S[k] = ↯;
19                 i++; k++;
20             else                          // skip binding if in alt but not in iter
21                 j++;
22
23         res ← res ∪ {(v, S)} ;
24
25 return res
```

maps to be joined are ordered the same.

These observations are exploited in Algorithm 5 to give a merge-join [102] style algorithm for the join of two sequence maps with disjoint edge cover that has linear time complexity in the (combined) size of the input sequence maps. Since the bindings are already in the same order, we can omit the sort phase of the merge join and immediately merge the two binding sequences. However, we need to ensure that not only the order but also the number of bindings (and the position of eventual failure markers, cf. lines 18–20) reflects that for the same variable $v$ in the sequence map where $v$'s incoming edge is in the edge cover (lines 9–11).

**Algorithm 6**: NextBinding(S, i)

**input** : Sequence $S$ containing, possibly, failure markers and start index $i$

**output**: The next element in $S$ at or after $i$ that is not a failure marker and its index or $(\infty, \infty)$ if no such binding exists

```
1 for j ← i to |S| do
2     if S[j] is not a failure marker then break;
3 if j = |S| and S[j] failure marker then return (∞, ∞) ;
4 return (S[j], j)
```

**Theorem 12.4.** *Algorithm 5 computes* $\dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$ *for sequence maps with disjoint edge cover and set of shared variables* Shared *in* $\mathcal{O}(b^{total}_{Shared} \cdot i) \leq \mathcal{O}(|Shared| \cdot n \cdot i)$ *time where* $b^{total}_{Shared}$ *is the total number of bindings associated in either sequence map with a variable in* Shared *and* $i$ *is the maximum number of intervals associated with any such binding. For tree, forest, and* CIG *data* $i = 1$*, for arbitrary graph data* $i \leq n$*.*

*Proof.* Algorithm 5 computes $S = \dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$: For any variable $v$, if a binding for $v$ occurs in the induced relation of both sequence maps, it occurs also in $S$ due to lines 15–17. If $v$'s incoming edge is in the edge cover of one of the sequence maps $S'$, lines 9–11 ensure that the sequence of bindings for $v$ is the same (except that some bindings are "bombed") in $S$ as in $S'$. For the parent $v'$ of $v$, if a binding is retained the set of intervals from both sequence maps are copied en block. There are only intervals in $S'$ (as $(v', v)$ is not in the edge cover of the other sequence map) and thus only those relations between bindings of $v$ and $v'$ as in the induced relation of $S'$ are retained. This is proper as in the induced relation of the other sequence map *all* bindings of $v$ are related to all bindings of $v'$ by definition of the induced relation. Both input sequences may be inconsistent: The presence of failure markers in either sequence does not

affect the correctness of the algorithm: failure markers in alt are skipped, failure markers in iter are retained (lines 15–17) as intended. Dangling bindings do not affect the algorithm.

Algorithm 5 loops over all shared variables of $S_1$ and $S_2$ and for each such variable it iterates over all bindings in the primary sequence map iter and corresponding bindings in alt, skipping, if necessary, bindings in alt not in iter. In the loop lines 13–22 $i$ or $j$ is incremented (possibly multiple times, if failure markers are skipped in NextBinding) until either $i > |\text{iter}|$. If $j$ ever becomes $> |\text{alt}|$ subsequent calls of $\text{binding}(()\text{alt}[j])$ return, by definition, a value larger than all $n \in \text{Nodes}(D)$.

Thus the algorithm touches, for each shared variable, each entry in either sequence map at most once (and touches one proper (not a failure mark) entry in each step of the loop 13–22). Thus it runs in $\mathcal{O}(b^{total}_{\text{Shared}} \cdot i)$ where $b^{total}_{\text{Shared}}$ is the total number of bindings in both sequence maps for a shared variable and $i$ is the maximum number of intervals per binding. This is bound by $\mathcal{O}(|\text{Shared}| \cdot n \cdot i)$ for any sequence map (including sequence maps for arbitrary graphs) as shown in Section 11.4. □

It is worth pointing out, that in a CIQCAG expression for a tree query any variable is shared at most once for each in- or outgoing edge and for each unary relation associated with the variable. Thus, even if there are $\mathcal{O}(q)$ joins in the expression, the accumulated number of shared variables among all those joins is also only $\mathcal{O}(q)$ an thus the complexity for only those joins is bounded by $\mathcal{O}(q \cdot n \cdot i)$.

*Consistent Join*

*Consistent sequence map*

In contrast to the map initialization, joining two sequence maps becomes considerably more complicated if we modify the definition to require that the resulting sequence map is *consistent*.

**Definition 12.4** (Consistent sequence map join (disjoint edge covers))**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two *consistent* sequence maps for $D$ over $Q$ such that *their associated edge covers are disjoint*. Then $\ddot{\bowtie}_\cap(S_1, S_2)$ returns a *consistent* sequence map $\overset{D,Q}{\text{sm}}_3$ that also fulfills all the conditions for $\ddot{\bowtie}^{\not z}_\cap(S_1, S_2)$.

When computing the consistent join sequence map, we have to adapt the interval referencing from one binding to a sequence of bindings of one of its child variables. This is necessary even if both input sequence maps are consistent, as some bindings may occur in one but not in the other sequence map and thus intervals shrink or even collapse.

We can use the previous algorithm for the possibly inconsistent variant as a starting point. However, we process now all variables that are either

shared or from which a shared variable is reachable by edges covered by the union of the edge covers of the input sequence map. This is necessary, as changes to one variable $v$ have to be propagated to its parent $v'$, as some of the parents bindings may reference only bindings for $v$ that are dropped. Such bindings for $v'$ must be dropped and, accordingly, may affect bindings for the parent of $v'$ and so on. These variables are processed in inverse topological order w.r.t. the edges in the edge covers of the two sequence maps, i.e., child variables before parent variables. For each variable, we record how the indices of bindings have changed: Let $i$ be an index for a binding $n$ of variable $v$ in one of the sequence map. Then, **(1)** if $n$ also occurs in the other sequence map, we retain $n$ and log $(v, \downarrow, i, k)$ where $k$ is the new index for $n$ and $\downarrow$ is the type of the log entry, here a mapping for a retained binding. **(2)** if $n$ does not occur in the other sequence map, we drop it and log both the index (in the original sequence of bindings for $v$) of the next retained entry (with type $\hookrightarrow$) and of the last preceding retained entry (with type $\hookleftarrow$).

With these change logs we can then adapt the intervals referring to bindings of $v$ when we process the parent variable for $v$ later (it comes after $v$ in inverse topological order).

Algorithm 7 gives the computation of the consistent join for sequence maps with disjoint edge covers and reflects these modifications from Algorithm 5. Note, that for each variable and index there is *either* a $\downarrow$ entry in the change log or both a $\hookleftarrow$ and a $\hookrightarrow$.

*Adapt intervals*

We use an additional helper function Adapt that is detailed in Algorithm 8 and actually applies a "change log" to a set of intervals: For each interval, we look at the start index. If it has a $\hookrightarrow$ entry (and thus the referenced binding is not retained) we set $s$ to the index in that entry, i.e., the index (in the original sequence of bindings) of the next retained binding. The same we do for the end index, but with its $\hookleftarrow$ entry, if there is any. After these adaptations, the start and end index might actually overlap if all bindings in the original interval have been dropped. Note, that the border cases are covered as we use $\infty$ to indicate that the start index is no "outside" the sequence and 0 to indicate that the end index is "outside", cf. line 21 and 24 in Algorithm 7 if the first element is bombed its $\hookleftarrow$ becomes 0 and line 26 if the last (or last few) element is bombed.

If that is not the case, the new start and end indices are retained and we can retrieve their $\downarrow$ entries with the indices in the new sequence. Finally, we ensure that intervals separated in the original sequence by entries dropped in the new sequence collapse (lines 9–12). This guarantees that the new set of intervals is actually minimal.

It is worth pointing out, that Adapt requires that the intervals are non-overlapping (as is guaranteed if they come from a sequence map, by defi-

**Algorithm 7:** $\ddot{\bowtie}_{\cap}(S_1, S_2)$

**input** : *Consistent* $S_1$ and $S_2$ with *disjoint* edge covers
**output**: *Consistent* sequence map representing the join of the input maps

```
1  EC1 ← edgeCover(S1); EC2 ← edgeCover(S2); Shared ← dom S1 ∩ dom S2;
2  PendingVars ← {v ∈ Vars(Q) : v ∈ Shared ∨ ∃ path from v to v' in EC1 ∪ EC2 with v' ∈ Shared};
3  res ← ∅; AdaptedVars ← ∅; Log ← {(S1, ∅), (S2, ∅)};
4  while PendingVars ≠ ∅ do
       // Select some variable v without children
5      v ∈ {v' ∈ PendingVars : ∄ v'' ∈ PendingVars : parent(v'') = v'} ;
       // last records the start of the current interval of dropped bindings, ∞ indicates that
          there is no such interval
6      last_I ← last_A ← ∞; iter ← S1; alt ← S2; S ← ∅; i, j, k ← 1 ;
7      if v ∉ dom S1 then iter ← S2; alt ← ⊥;
8      if v ∉ dom S2 then alt ← ⊥;
9      while i ≤ |iter(v)| do
10         I ← ∅; n1 ← binding(iter(v)[i]);
11         n2 ← binding(alt(v)[j]) or n1 if alt = ⊥;
           // Compute the adapted intervals (if necessary) and "bomb" entry if all
              intervals dropped
12         if (v, v') ∈ EC1 ∪ EC2 for some variable v' then
13             I1 ← intervals(iter(v)[i]); I2 ← intervals(alt(v)[j]) or ∅ if alt = ⊥;
14             I ← Adapt(I1, Log(iter)) ∪ Adapt(I2, Log(alt));
15             if I = ∅ then n1 ← o
16         if n1 = n2 then                                   // retain binding if same
17             S[k] ← (n1, I);
               // Log: i changes to k; i next good binding for all "bombed" bindings
                  directly before i (if there are none last_i = ∞)
18             Log(iter) ← Log(iter) ∪ {(v, ↓, i, k)} ∪ {(v, ↪, l, i) : last_I ≤ l < i}
               Log(alt) ← Log(alt) ∪ {(v, ↓, j, k)} ∪ {(v, ↪, l, j) : last_A ≤ l < j} i++;
               j++; k++; last_I ← last_A ← ∞;                // incr. counters; reset last
19         else if n1 < n2 then                              // skip binding if in iter but not in alt
20             last_I ← min(last_I, i);                      // start of current interval of only "bombs"
               // Record the last "good" binding before i
21             Log(iter) ← Log(iter) ∪ {(v, ↩, i, last_I − 1)}; i++;
22         else                                              // skip binding if in alt but not in iter
23             last_A ← min(last_A, j) ;
24             Log(alt) ← Log(alt) ∪ {(v, ↩, j, last_A − 1)}; j++;
25
26     set next "good" binding for open "bombed" intervals to ∞ in Log;
27     PendingVars ← PendingVars ∖ {v}; AdaptedVars ← AdaptedVars ∪ {v};
28     res ← res ∪ {(v, S)};
29 copy bindings for variables not in AdaptedVars to res
30 return res
```

nition of a sequence map). The generated intervals are again overlapping for log entries resulting from Algorithm 7.

For example for the interval set $\{(v,3,5),(v,7,7),(v,10,12)\}$ over a sequence from 1 to 12 and the log entries $\{(v,\hookrightarrow,3,4),(v,\hookleftarrow,3,0),(v,\downarrow,4,2),(c,\hookrightarrow,5,7),(v,\hookleftarrow,5,4),(v,\downarrow,7,3),(v,\hookrightarrow,8,9),(v,\hookleftarrow,8,7),(v,\downarrow,9,4),(v,\hookrightarrow,\infty),(v,\hookleftarrow,10,9),(v,\hookrightarrow,12,\infty),(v,\hookleftarrow,12,9),\ldots\}$ (the log is larger but the remaining entries are not of interest). From the log, we can conclude that only the bindings for $4,7,9$ are retained. The interval $[3,5]$ collapses to the interval $[1,1]$ (the old 4 is now the 1st binding) and then collapses with $[7,7]$ that is now $[2,2]$. The interval $[10,12]$ is dropped entirely. Thus the adapted set of intervals is $\{[1,2]\}$ (note that 9, now 3, is not covered by the intervals in this set).

**Theorem 12.5.** *Algorithm 8 computes the adapted set of intervals for a given set of intervals $I$ and a change log $L$ in $\mathcal{O}(|I|)$ time assuming constant membership test in $L$.*

*Proof.* Note, that $L$ contains for each pair $(v,i)$ either one $\downarrow$ or one $\hookleftarrow$ and one $\hookrightarrow$ entry. Thus $L$'s size is bounded by $\mathcal{O}(q \cdot n)$ where for a given sequence map $\overset{D,Q}{\mathsf{sm}}$ and constant membership test can be realized by an array over variables and indices, each cell containing either the $\downarrow$ or the $\hookleftarrow$ and $\hookrightarrow$ entry, with the same space bound.

The iteration over the intervals in order of start indices (line 4) is linear, since the intervals are stored ordered, cf. Section 12.2.1.

The algorithm iterates over all intervals in $I$ and for each interval performs a series of membership tests in $L$ or simply copies the interval if there are no change entries for the involved variable. □

*Comparison of the two approaches*

The advantage of $\ddot{\bowtie}_\cap()$ vs. $\ddot{\bowtie}^{\not\downarrow}_\cap()$ is quite obvious: we get a consistent and thus in most cases smaller resulting sequence map. The sequence map computed by $\ddot{\bowtie}^{\not\downarrow}_\cap()$, on the other hand, most likely contains some redundancies. This effect is all the more pronounced the more selective the involved sequence maps are.

However, the computation of $\ddot{\bowtie}_\cap()$ is also significantly more involved than that of $\ddot{\bowtie}^{\not\downarrow}_\cap()$. It requires $\mathcal{O}(|\mathcal{V}| \cdot n)$ additional space for the change log. $\mathcal{V}$ is the set of variables that are either shared or from which a shared variable is reachable by edges in the union of the edge covers of the input sequence maps (PendingVars in line 3). Where $\ddot{\bowtie}^{\not\downarrow}_\cap()$ ignores non-shared variables (and can copy all intervals en-block), $\ddot{\bowtie}_\cap()$ must touch all shared variables and their ancestors. For each of these, it iterates over all its associated intervals to adapt their start and end index. Thus, the total number of intervals associated to any binding for a variable in $\mathcal{V}$ forms a lower bound for $\ddot{\bowtie}_\cap()$. Algorithm 7 runs in $\mathcal{O}(b^{total}_{\mathcal{V}} \cdot i)$ where $b^{total}_{\mathcal{V}}$ is the total

**Algorithm 8**: Adapt(Ints, Log)

**input** : Non-overlapping set of intervals Ints and set of "log" entries Log as generated from Algorithm 7

**output**: Set of intervals modified according to Log

```
1  NewInts ← ∅ ;
   // For variables with entries in the "Log": adapt
2  foreach v ∈ π₁(Log) do
3      lastStart ← ∞; lastEnd ← ∞;
4      foreach (v, s, ∈ Ints in order of start index do
5          if (v, ↪, s, next) ∈ Log then s ← next;
6          if (v, ↩, e, prev) ∈ Log then e ← prev;
7          if s > e then continue;
           // Now there must be ↓ entries for s and e, as ↪ and ↩ entries
              always reference "good" entries or ∞ or ○ which are
              excluded above
8          (v, ↓, s, s_new), (v, ↓, e, e_new) ∈ Log ;
           // If we cannot extend the last interval, add it …
9          if lastEnd ≠ s_new + 1 then
10             NewInts ← NewInts ∪ {(v, lastStart, lastEnd)};
11             lastStart = s_new ;
           // … otherwise lastStart remains unchanged
12         lastEnd ← e_new ;
       // Collect remaining interval
13     if lastStart ≠ ∞ then NewInts ← NewInts ∪ {(v, lastStart, lastEnd)};
   // For variables without entries in the "Log": copy
14 foreach (v, s, e) ∈ Ints with v ∉ π₁(Log) do
15     NewInts ← {(v, s, e)};
16 return NewInts
```

number of bindings for all variables in $\mathcal{V}$ in both sequence maps and $i$ the maximal number of intervals per binding. This is bounded by $\mathcal{O}(q \cdot n \cdot i)$ where $i$ is the maximal number of intervals per binding. For tree, forest, and CIG queries $i = 1$, for arbitrary queries $i \leq n$, cf. Section 11.4.

**Theorem 12.6.** *Algorithm 7 computes $\ddot{\bowtie}_{\cap}(S_1, S_2)$ for sequence maps with disjoint edge cover in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot i) \leq \mathcal{O}(q \cdot n \cdot i)$ time where $i$ is the maximal number of intervals per binding and $b_{\mathcal{V}}^{total}$ is the total number of bindings for shared variables or variables from which shared variables are reachable by edges in the union of the edge covers of the two sequence maps, $n = |Nodes(D)|$ and $q = |Vars(Q)|$. It requires $\mathcal{O}(|V| \cdot n)$ additional space for the change log.*

*Proof.* Algorithm 7 computes $S = \ddot{\bowtie}_{\cap}(S_1, S_2)$: A tuple $t \in R_{S_1} \bowtie R_{S_2}$, iff for all variables $v$ with $t[v] = n$ **(1)** $n$ occurs among the bindings of $v$ in both sequence maps and thus among the bindings of $v$ in the resulting sequence map (lines 16–18) and, **(2)** for all variables $v'$ with $(v', v)$ in the edge cover of either sequence map, the interval pointer in all bindings for $v'$ are adapted using the change log (note that only one of sequence maps contains bindings for $v'$ with interval pointers to $v$ since the edge covers are disjoint. The change log contains a mapping from each index $i$ of $S_1$ to an index $k$ in the result sequence map, if the binding with index $i$ is retained. Otherwise, it references back and forward to the last previous and next following index of a binding that is not dropped. Adapt adapts the old intervals to the new indices. If all bindings in an interval are dropped, the interval is removed. If all bindings for $v$ in all intervals for a binding $n$ of $v'$ are dropped (and thus, one of the sequence maps contains no tuples with bindings for $v$ in any of the intervals for $n$ and thus no tuple containing $n$), $n$ is eliminated (cf. line 15 and 19–21).

The result of Algorithm 7 is consistent: There are no failure markers as the algorithm does not introduce any (and the input sequence maps are consistent). If there are no dangling bindings in the original sequence maps, then there are no dangling bindings in the resulting sequence map, as any binding that is retained and is covered in the original sequence map is also covered in the new sequence map by the construction of the adapted intervals (lines 17–18 and lines 21 and 24).

The algorithm loops over the variables in AllVars, each iteration removing one variable until AllVars is empty. For the inner loop reasoning analog to Algorithm 5 applies except if the binding is retained: Then we also call Adapt which requires time linear in the size of the intervals.

As Algorithm 5, the algorithm touches each entry in either sequence map at most once (and touches one such entry in each step of the loop 13–25), but for each entry it can not copy all intervals en block, but needs

to adapt each interval. Thus it runs in $\mathcal{O}((b_{S_1} + b_{S_2}) \cdot i)$ where $b_{\mathsf{sm}}^{D,Q}$ is the total number of bindings in a sequence map and $i$ the maximal number of intervals per binding. This is bound by $\mathcal{O}(q \cdot n \cdot i)$ for any sequence map (including sequence maps for arbitrary graphs) as shown in Section 11.4.

Regarding the space bound, see AdaptIntervals and the proof of Theorem 12.5. □

Note that the algorithm actually handles and even corrects failure markers in the input sequence maps, but does not allow dangling bindings if the result is expected to be consistent. In other words, we could weaken the requirement that both input sequence maps are consistent to only that they do not contain dangling bindings.

Though, at the first glance, the time bounds for the two algorithms are similar, the above observation that variables in CIQCAG expressions for tree queries are shared at most once per edge and at most once per associated unary relation, does not hold. Rather, variables with many descendants in the query are likely to be considered in many joins, in particular if the constraints of a query a enforced in bottom-up fashion. In general, this means that for a CIQCAG expression with $q$ joins evaluating a tree query we incur a total cost of $\mathcal{O}(q \cdot q \cdot n \cdot i)$ for the processing of all joins. This is an increase by a multiplicative factor $q$ compared to the use of $\dddot{\bowtie}^{\not{\iota}}_{\cap}()$.

If we compare this result with the combination of the possibly inconsistent $\dddot{\bowtie}^{\not{\iota}}_{\cap}()$ with the propagate operator (cf. Section 12.5.3) we can observe the essential advantage of allowing (temporarily) inconsistent sequence maps: If we use explicit propagate we can touch and adapt the intervals of each variable once (after all restrictions for that variable and its descendant variables have been evaluated). With implicit propagate, we have to potentially "touch" them in each join and thus introduce an additional multiplicative factor in the order of the size of the query. For details, see Section 12.5.3.

For the join, as for many of the operators defined in the following, there is a variant for consistent and a variant for inconsistent variants, the prior requiring consistent sequence maps and returning consistent sequence maps, the latter allowing any sequence map and returning sequence maps with possible inconsistencies. All operators for consistent sequence *preserve interval-minimality*, i.e., if given interval-minimal, consistent sequence maps as input, they produce interval minimal consistent sequence maps. The propagation operator actually ensure that the resulting sequence map is *interval-minimal*.

| $v_1$ | $v_2$ | $v_3$ |
|---|---|---|
| $d_1$ | $d_3$ | $d_4$ |

| $v_1$ | $v_2$ | $v_3$ |
|---|---|---|
| $d_1$ | $d_3$ | $d_5$ |
| $d_2$ | $d_3$ | $d_4$ |

Figure 62. Relations $R_1$ and $R_2$ where join on decomposed relations seems insufficient

*General Join*

*Relaxing the edge cover restriction*

The second variation of the sequence map join revolves around the restriction to sequence maps with disjoint edge covers. What is the effect if we allow the edge covers to overlap?

If we relax the edge cover restriction, we allow that both sequence maps represent some subset of the possible combinations of bindings for, e.g., the three variables $v_1, v_2, v_3$. At first glance, this seems to make it impossible to join such sequence maps and represent the result as a sequence map (rather than a relation where we track combinations of all variables rather than only references from $v_1$ to $v_2$ and, separately, from $v_2$ to $v_3$). Figure 62 shows two such relations where we can not join separately: If we consider only references from $v_1$ to $v_2$, we retain the pair $(d_1, d_3)$. If we consider only references from $v_2$ to $v_3$, we retain the pair $(d_3, d_4)$ since both pairs occur in each relation. Thus the resulting sequence map represents the relation $\{(d_1, d_3, d_4)\}$ which is different from $R_1 \bowtie R_2$.

However, the reason for this behavior is that $R_2$ from Figure 62 can actually not be represented as a sequence map (it does not exhibit a lossless-join decomposition into binary relations over the pairs of adjacent attributes): Should we reference $d_4$, $d_5$, or both from $d_3$'s binding for $v_2$? In contrast to $R_2$, all relations representable as a sequence map actually ensure that if there is any tuple with $(n, n')$ as bindings for $v_2, v_3$ then, for any combination of values for the remaining variables (here $v_1$), there must be a tuple with $(n, n')$ as bindings for $v_2, v_3$, cf. Section 11.2. In our example $R_2$ must be extended with the tuples $(d_1, d_3, d_4)$ and $(d_2, d_3, d_5)$ to be amenable for representation as sequence map.

This property is what makes it possible to define a variant of the sequence map join that allows overlapping edge cover yet still computes the join for each adjacent pair of variables separately. Thus we can define the general sequence map join as follows:

*Computing the general sequence map join*

**Definition 12.5** (General sequence map join)**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two *arbitrary* (consistent) sequence maps for $D$ over $Q$. Then $\ddot{\bowtie}^{\lightning}(S_1, S_2)$ ($\ddot{\bowtie}(S_1, S_2)$returns a (consistent) sequence

map $\overset{D,Q}{\mathbf{sm}}_3$ that also fulfills all the conditions for $\ddot{\bowtie}^{\not\cap}_{\cap}(S_1, S_2)$.

Algorithms for this join variant can be fairly easily derived from the Algorithms for $\ddot{\bowtie}^{\not\cap}_{\cap}$ and $\ddot{\bowtie}_{\cap}$: The previous algorithms for a sequence map join use a union (line 16 in Algorithm 5, line 17 in Algorithm 7) to combine intervals from both sequence maps, since we know that one of the two sets of intervals is always empty (otherwise the edge covers overlap) and thus no further "joining" is necessary. For the general sequence map join, *both* sequence maps may contain interval pointers for the same edge and we have to ensure that only those references are retained for that an interval pointer exists in both input sequence maps.

*Joining intervals*

To compute $\ddot{\bowtie}(S_1, S_2)$, we modify line 14 in Algorithm 7 to use a new function JoinInts, instead of $\cup$ to combine the two adapted sets of intervals. This function is defined in Algorithm 9. First, for variables covered in both interval sets, we iterate (lines 3–16) in parallel over the two sets of intervals (that are non-overlapping and thus can be ordered, e.g., by start index). For each interval, we look obtain the next intervals of the other set as long as there is an overlapping or the interval is entirely to the right. In the latter case, we take another interval from the first set etc. For variables covered only in one interval set, we simply retain the existing intervals. With this adaptation, we obtain an algorithm for $\ddot{\bowtie}()$.

Note, that JoinInts is surprisingly simple as both sets contain non-overlapping intervals. Thus a simple traversal in the order of the start (or, equivalently, end) indices is possible. An alternative is the use of an interval tree or, since our intervals are non-overlapping, even of B-trees indexing start and end intervals. Though interval trees are, in general, very efficient (logarithmic) at answering point or stab queries (where we give an index and retrieve all intervals containing that index), we can exploit here that we are interested in the full interval cover instead of a single point query. Realizing JoinInts as a sequence of point queries (for all covered indices of a child relation) is possible, but comes at a higher complexity (by the logarithmic look-up time factor) than the above algorithm.

The algorithm touches each interval in both sets $A$ and $B$ exactly once. Each interval in $A$ may be compared to several intervals of $B$, though all but the last of these are not compared to any other from interval from $A$. In total, this gives a bound of $\mathcal{O}(|A| + |B|)$. We assume that the access to the intervals in order of the start indices (lines 4–5) is constant for each interval and linear for the entire set. In other words, we assume that the intervals are stored ordered by the start index (cf. Section 12.2.1).

**Theorem 12.7.** *Algorithm 9 computes the join over the set of non-overlapping intervals $I_1$ and $I_2$ in $\mathcal{O}(|I_1| + |I_2|)$ time.*

**Algorithm 9**: JoinInts($\text{Intervals}_1$, $\text{Intervals}_2$)

**input** : Two sets $\text{Intervals}_1$, $\text{Intervals}_2$ of non-overlapping intervals with associated variables

**output**: Minimal, non-overlapping set of intervals covering all indices contained in intervals of both sets

```
1  NewIntervals ← ∅ ;
2  start ← ⊥ ;
3  foreach v ∈ π1(Intervals1) ∩ π1(Intervals2) do
4      (v, s1, e1) ← interval in Intervals1 with minimal start index s1;
5      (v, s2, e2) ← interval in Intervals2 with minimal start index s2;
6      while true do
7          while e2 < s1 do
               // Current (s2, e2) do not overlap, get the next
8              Intervals2 ← Intervals2 ∖ {(v, s2, e2)};
9              if Intervals2 = ∅ then break 2 (line 6);
10             (v, s2, e2) ← interval in Intervals2 with minimal start index
                 s2;
           // There is some overlapping
11         if s2 ≤ e1 then
12             NewIntervals ←
                 NewIntervals ∪ {(v, max(s1, s2), min(e1, e2))};
13         if e1 ≤ e2 or s2 > e1 then
               // Current (s1, e1) do not overlap (any further), get the
                  next
14             Intervals1 ← Intervals1 ∖ {(v, s1, e1)};
15             if Intervals1 = ∅ then break (line 6);
16             (v, s1, e1) ← interval in Intervals1 with minimal start index
                 s1;
17 foreach v ∈ π1(Intervals1) ∖ π1(Intervals2) do
18     NewIntervals ← NewIntervals ∪ {(v, s, e) ∈ Intervals1};
19 foreach v ∈ π1(Intervals2) ∖ π1(Intervals1) do
20     NewIntervals ← NewIntervals ∪ {(v, s, e) ∈ Intervals2};
21 return NewIntervals ;
```

Thus, the use of join interval does not significantly affect the complexity of Algorithm 7:

**Theorem 12.8.** *Algorithm 7 with* JoinInts *instead of* $\cup$ *in line 14, computes* $\ddot{\bowtie}_{\cap}(S_1, S_2)$ *for sequence maps with disjoint edge cover in* $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot i) \leq \mathcal{O}(q \cdot n \cdot i)$ *time where* $i$ *is the maximal number of intervals per binding and* $b_{\mathcal{V}}^{total}$ *is the total number of bindings for shared variables or variables from which shared variables are reachable by edges in the union of the edge covers of the two sequence maps,* $n = |\mathit{Nodes}(D)|$ *and* $q = |\mathit{Vars}(Q)|$. *It requires* $\mathcal{O}(|V| \cdot n)$ *additional space for the change log.*

*Proof.* The modified algorithm ensures that the resulting sequence map contains a reference from a binding $n$ for variable $v$ to a binding $n'$ for a variable $v'$ if and only if both input sequence maps contain such a reference.

In addition to the argument $\ddot{\bowtie}_{\cap}$, we can observe one more case: a pair of variables $v, v'$ is in the edge cover of *both* sequence maps. In this case, any tuple $t$ in the induced relation of either sequence map with $t[v] = n$ and $t[v'] = n'$ implies that there is an interval pointer in the set associated with $n$ among the bindings for $v$ that covers the index of $n'$ in the bindings of $v'$. If $t$ is in both sequence maps, such pointers exists in both of them and $t$ is also in the join of the two induced relations. Then JoinInts ensures that there is an interval pointers in the new sequence map from $n$ to an interval containing the index of $n'$ among the bindings for $v'$. With the observation that the same holds for every pair of variables, we obtain that $t$ is in the induced relation of the resulting sequence map.

The complexity follows from Theorems 12.6 and 12.7. □

*Inconsistent general join*

For the variant that accepts possibly inconsistent sequence maps as result, denoted with $\ddot{\bowtie}^{\lightning}(S_1, S_2)$, the modifications are a bit more extensive and move the inconsistent variant closer to the consistent one: we need to introduce change tracking for the secondary sequence (alt in Algorithm 5) and use those changes to adapt the intervals in alt so that we can join them with the intervals from iter (that, as before, do not need to be adapted). However, as for Algorithm 5 and in contrast to the consistent case, only intervals of shared variables and their direct parents, but not of other ancestor variables, must be adopted. The the move from union to JoinInts does not affect the cost for a single variable, however, we also need to cover direct parent variables. Since each shared variable has a unique parent (if any), this, nevertheless, increases the required time only a constant multiplicative factor.

*Sequence maps are closed under join*

Since we are now able to compute the join of two arbitrary sequence maps such that the induced relation of the resulting sequence map is the natural join of the induced relation of the input sequence maps, we can conclude the following theorem:

**Theorem 12.9.** *For any two sequence maps $S_1, S_2$, the join of their induced relations can be represented as a sequence map.*

As discussed above, the reason this statement holds is that the induced relations of the input sequence maps themselves can be represented in a sequence map and thus have lossless-join decompositions to binary relations over each pair of adjacent variables. Therefore, either a particular binding pair for two adjacent variables is eliminated from *all* binding tuples (if it is only in one of the sequence maps) or from *none*. In both cases, the resulting relation still is fully decomposable and thus can be represented by a sequence map.

*Semi-Join*

As a convenience, we also introduce a specific (left) semi-join operator $\dddot{\ltimes}^{\lightning}(S_1, S_2)$. The definitions and algorithms can be easily derived from the join operator and are only sketched here. First we adopt the (possibly inconsistent) definition of $\dddot{\bowtie}_{\cap}()$:

**Definition 12.6** (Sequence map semi-join (disjoint edge covers))**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two sequence maps for $D$ over $Q$ such that *their associated edge covers are disjoint* and $S_2$ is a single-variable sequence map. Then $\dddot{\ltimes}^{\lightning}(S_1, S_2)$ returns a sequence map $\overset{D,Q}{\mathsf{sm}}_3$ such that

**(1)** the induced relation of $\overset{D,Q}{\mathsf{sm}}_3$ is the left semi-join of the induced relations of $S_1$ and $S_2$, i.e., $R_{\overset{D,Q}{\mathsf{sm}}_3} = R_{S_1} \ltimes R_{S_2}$.

**(2)** $\overset{D,Q}{\mathsf{sm}}_3|_{\text{dom } S_1 \cup \text{dom } S_2} = \overset{D,Q}{\mathsf{sm}}_3$ (contains bindings only for variables mapped either in $S_1$ or in $S_2$).

The associated edge cover for $\dddot{\ltimes}^{\lightning}(S_1, S_2)$ is the edge covers associated with $S_1$.

We limit the second sequence map to a single-variable sequence map to avoid cases, where the resulting relation no longer exhibits a full join decomposition and can thus not be represented in a sequence map. Thus, the semi-join operators is often combined with a projection (see Section 12.5.1) to a single variable.

We can compute $\dddot{\ltimes}^{\lightning}$ by a slightly modified version of Algorithm 5: We drop line 7 and thus do not retain bindings for variables only in the second sequence map. Furthermore, in line 15 we only retain intervals from iter not from alt (i.e., we drop the second operand from the union). The resulting algorithm computes $\dddot{\ltimes}^{\lightning}$ with the same complexity as Algorithm 5.

An analog modification for Algorithm 7 yields a consistent sequence map (we copy only bindings from the first sequence map in line 26 and

drop $I_2$ and all its references in lines 12–15. Again, the change does not affect significantly the complexity of the algorithm.

### 12.4.2 UNION

Where the sequence map join requires that only bindings of shared variables that are contained in both input sequence maps are retained, the sequence map union, denoted by $\ddot{\cup}()$, accepts bindings contained in either sequence map.

*Union sequence maps*

The union (and difference) of sequence maps is defined in the following only for single-variable or single-edge sequence maps. A sequence map $\overset{D,Q}{\mathbf{sm}}$ over a relational structure $D$ and a tree query $Q$ is called *single-variable*, iff $|\text{dom } SM| = 1$. It is called *single-edge*, iff $\text{dom } \overset{D,Q}{\mathbf{sm}} = \{v, v'\}$ and $\text{edgeCover}(\overset{D,Q}{\mathbf{sm}}) = \{(v, v')\}$, i.e., if there is a single edge in the edge cover of $\overset{D,Q}{\mathbf{sm}}$ and the only variables mapped by $\overset{D,Q}{\mathbf{sm}}$ are that edge's source and sink. This restriction allows that union is well-defined and closed over such sequence maps and in accordance to the union over the induced relations of their input sequence maps.

**Definition 12.7** (Sequence map union)**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two single-variable sequence maps or two single-edge sequence maps for $D$ over $Q$ with $\text{dom } S_1 = \text{dom } S_2$ and the same associated edge cover (empty if single-variable, the same single edge if single-edge). Then $\ddot{\cup}(S_1, S_2)$ returns a (single-variable or single-edge) sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that

**(1)** the induced relation $R_{\overset{D,Q}{\mathbf{sm}}_3}$ of $\overset{D,Q}{\mathbf{sm}}_3$ is $R_{\overset{D,Q}{\mathbf{sm}}_3} = R_{S_1} \cup R_{S_2}$, i.e., the union of the induced relations of $S_1$ and $S_2$.

**(2)** $\overset{D,Q}{\mathbf{sm}}_3|_{\text{dom } S_1 \cup \text{dom } S_2} = \overset{D,Q}{\mathbf{sm}}_3$ (contains bindings only for variables mapped either in $S_1$ or in $S_2$).

The associated edge cover for $\ddot{\cup}(S_1, S_2)$ is the edge cover associated with $S_1$ and $S_2$, i.e., either empty if both input sequence maps are single-variable or the single edge in their edge cover.

The limitation to single-variable and single-edge sequence maps mirrors the restrictions for relational union (same schema for both relations). However, the limitation is necessarily stronger, as discussed above.

Algorithm 10 gives an implementation of this operator. Notice, the similarity and differences to Algorithm 7: Though the main skeleton is similar, we actually retain bindings in each of the cases in lines 11–22, only failure markers are skipped (line 10). We also only record $\downarrow$ mappings from old indices to new ones, but no $\hookleftarrow$ and $\hookrightarrow$ mappings.

If both input maps are single-edge, the child variable is processed first (by means of line 5) and, for each index of any binding (except for failure

**Algorithm 10**: $\ddot{\cup}(S_1, S_2)$

**input** : Single-variable or single-edge Sequence maps $S_1$ and $S_2$ as in Definition 12.7

**output**: Sequence map res representing the union of the induced relations of the input maps

```
1  SharedVars ← dom S₁ (= dom S₁ ∩ dom S₂) ;
2  EC ← edgeCover(S₁) (= edgeCover(S₂));
3  Log ← {(S₁ ↦ ∅), (S₂ ↦ ∅)}; res ← ∅;
4  while SharedVars ≠ ∅ do
5      v ∈ {v' ∈ SharedVars : ∄ v'' ∈ SharedVars : (v'', v') ∈ EC} ;
6      S ← ∅; i, j, k ← 1;
7      while i ≤ |S₁(v)| or j ≤ |S₂(v)| do
           // bindings returns ∞ for index out of bound
8          n₁ ← binding(S₁(v)[i]); n₂ ← binding(S₂(v)[j]);
           // intervals returns ∅ for index out of bound
9          I ←
           RecreateInts(S, intervals(S₁(v)[j]), intervals(S₂(v)[j]), Log);
10         if n₁ = ↯ then i++ else if n₂ = ↯ then j++
11         else if n₁ = n₂ then
12             Log(S₁) ← Log(S₁) ∪ {(v, ↓, i, k)};
13             Log(S₂) ← Log(S₂) ∪ {(v, ↓, j, k)};
14             i++; j++; k++;
15         else if n₁ < n₂ then
16             S[k] ← (n₁, I);
17             Log(S₁) ← Log(S₁) ∪ {(v, ↓, i, k)};
18             i++; k++;
19         else if then
20             S[k] ← (n₂, I);
21             Log(S₂) ← Log(S₂) ∪ {(v, ↓, j, k)};
22             j++; k++;
23     res ← res ∪ {(v, S)};
24     SharedVars ← SharedVars ∖ {v};
25 return res
```

markers) for both sequence maps a mapping to the new index is established. Note, that all interval sets are empty and thus line 9 has no effect. Second, the parent variable is processed using the change log for the child variable. Now, there are interval sets pointing to bindings of the child variable and these intervals are adapted to the new indices in line 9 by means of RecreateInts. Algorithm 11 shows how RecreateInts is realized: For each variable covered by the passed interval sets (if called from Algorithm 10, there is always only a single such variable), we recreate the intervals. We can not simply adapt the intervals as in Adapt, Algorithm 8 (which is limited by the number of intervals in the input set), since within the range of an interval additional bindings (from the "opposite" sequence map) are introduce that are not actually mapped to that parent binding. This is impossible in the case of join and difference, where we only *restrict* but do not *extent* the bindings contained in each of the input sequence maps.

**Algorithm 11**: RecreateInts($S, I_1, I_2$, Log)

**input** : Sequence map $S$, *non-overlapping* interval sets $I_1, I_2$, and change log Log
**output**: Set of intervals modified according to Log

```
1  NewInts ← ∅ ;
   // For variables with entries in either interval set ...
2  foreach v ∈ π1(I1) ∪ π1(I2) do
3      i ← j ← k ← 1; start ← ∞;
4      for k ← 1 to |S(v)| do
5          n ← S(v)[k];
6          (v, ↓, idx1, k) ∈ Log or idx1 ← 0;
7          (v, ↓, idx2, k) ∈ Log or idx2 ← 0;
8          covered ← false;
9          if FallsIn(idx1, π2,3(I1)) then
10             I1 ← FallsIn(idx1, π2,3(I1)); covered ← true;
11         if FallsIn(idx2, π2,3(I2)) then
12             I2 ← FallsIn(idx2, π2,3(I2)); covered ← true;
13         if covered = true and start = ∞ then start ← k;
14         else if covered = false and start ≠ ∞ then
15             NewInts ← NewInts ∪ {(v, start, k − 1)};
16             start ← ∞;
       // Collect remaining interval
17     if start ≠ ∞ then NewInts ← NewInts ∪ {(v, start, |S(v)|)};
18 return NewInts
```

**Algorithm 12**: FallsIn(index, $I$)

**input** : Index idx and *non-overlapping* interval set $I$
**output**: The interval set $I'$ containing only those intervals from $I$ that cover or follow idx, or **false** if idx not covered by $I$

```
1 foreach (s, e) ∈ I in increasing order on start index s do
2     if idx < s then break;
3     if idx ≤ e then return I ;
4     I ← I ∪ {(s, e)} ;
  // idx is not covered by any interval in I
5 return false;
```

Depending on whether the input sequence maps are consistent (contain dangling bindings), we obtain the following result:

**Theorem 12.10.** *Algorithm 11 recreates a minimal set of non-overlapping interval pointers to bindings in a given sequence map S such that the index of each binding covered by an interval in that set is mapped by a given change log Log to an index covered in the set of intervals $I_1$ or the set of intervals $I_2$. It computes this set in $\mathcal{O}(b_V^{total} + |I_1| + |I_2|)$ where $b_V^{total}$ is the total number of bindings for variables occurring in $I_1$ or $I_2$, if the input sequence maps contain no dangling bindings, $\mathcal{O}(b_V^{total} \cdot (|I_1| + |I_2|))$ otherwise.*

*Proof.* The set of generated intervals is minimal and non-overlapping as the next index after any end index is either out-of-bound (line 15) or not covered by any interval (line 12–14) and the previous index before any start index is either out-of-bound or not covered by any interval (an interval starts only in line 11 and only if start is $\infty$ which is only the case at the beginning of the loop, line 3, or if the preceding index is not covered, line 12–14). Thus, we can expand neither interval. Also all intervals are by construction non-overlapping.

For each index $k$ that lies in an interval, there is an old index $\mathsf{idx}_1$ (or $\mathsf{idx}_2$) such that the old index lies in one of the intervals of $I_1$ (or $I_2$), line 9–11. Otherwise, it is not covered by an interval, line 12–14.

For the complexity, consider the loop 4–15 over the bindings for any variable in $\pi_1(I_1) \cup \pi_2(I_2)$. For each iteration, we call FallsIn which returns either the interval covering the index (and removes all previous intervals) or **false**.

If the underlying sequence map contains no dangling bindings each binding is covered by some index and the total run-time is bound by $\mathcal{O}(b_V^{total} + |I_1| + |I_2|)$ as each call to FallsIn runs in amortised constant time. If there are dangling bindings, however, for those bindings there is an

associated old index, but this old index is not covered by any interval. Thus, in worst case, FallsIn runs in $|I|$ for each call. □

If both input maps are single-variable, we can actually eliminate the change log and all references to it. We can also eliminate line 9 entirely, though RecreateInts anyway returns immediately in this case as the interval sets are always empty.

With this observation the following corollary follows directly from Theorem 12.4 and Theorem 12.10:

**Corollary 12.2.** *Algorithm 10 computes* $\ddot{\cup}(S_1, S_2)$ *for consistent single-variable or single-edge sequence maps with shared domain* Shared *and edge cover in* $\mathcal{O}(b_{\text{Shared}} \cdot b_{\text{Shared}}) \leq \mathcal{O}(|\text{Shared}| \cdot n^2)$ *time where* $b_{\text{Shared}}$ *is the maximum number of bindings associated in either sequence map with a variable in* Shared. *If either input map is inconsistent* $\mathcal{O}(b_{\text{Shared}} \cdot b_{\text{Shared}} + b^{dangle}_{\text{Shared}} \cdot b^{dangle}_{\text{Shared}} \cdot i) \leq \mathcal{O}(|\text{Shared}| \cdot n^2 \cdot i)$ *where* $b^{dangle}_{\text{Shared}}$ *is the maximum number of dangling bindings associated in either sequence map with a variable in* Shared, $i$ *is the maximum number of intervals associated with any such binding. For tree, forest, and* CIG *data* $i = 1$, *for arbitrary graph data* $i \leq n$.

There is no specific version that generates consistent sequence maps, as the above algorithm ensure that the resulting sequence map contains no failure markers if neither input sequence maps contained any (for shared variables, failure markers are actually removed due to NextBinding, for others the existing bindings are copied in line 3–4). Furthermore, if neither input sequence maps contains dangling bindings, no such bindings are introduced (all interval pointers remain untouched as do the binding sequences they reference due to the second condition imposed on the input sequence maps in Definition 12.7.

### 12.4.3 DIFFERENCE

The final combination operation for the sequence map removes bindings from a given sequence map if they are also contained in another sequence map. We pose much the same restrictions as on the sequence maps serving as input for the union operator.

**Definition 12.8** (Sequence map difference)**.** Let $D$ be a relational structure, $Q$ a tree query, and $S_1, S_2$ two single-variable sequence maps or two single-edge sequence maps for $D$ over $Q$ with $\operatorname{dom} S_1 = \operatorname{dom} S_2$ and the same associated edge cover (empty if single-variable, the same single edge if single-edge). Then $\ddot{\setminus}(S_1, S_2)$ returns a (single-variable or single-edge) sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that

**(1)** the induced relation $R_{\mathsf{sm}_3^{D,Q}}$ of $\mathsf{sm}_3^{D,Q}$ is $R_{\mathsf{sm}_3^{D,Q}} = R_{S_1} \setminus R_{S_2}$, i.e., the difference of the induced relations of $S_1$ and $S_2$.

**(2)** $\mathsf{sm}_3^{D,Q}|_{\mathrm{dom}\, S_1 \cup \mathrm{dom}\, S_2} = \mathsf{sm}_3^{D,Q}$ (contains bindings only for variables mapped either in $S_1$ or in $S_2$).

The associated edge cover for $\dddot{\setminus}(S_1, S_2)$ is the edge cover associated with $S_1$ and $S_2$, i.e., either empty if both input sequence maps are single-variable or the single edge in their edge cover.

$\dddot{\setminus}$ can be computed by an algorithm similar to the one for inconsistent sequence map join with the same time and space properties, cf. Algorithm 13.

It uses a helper function DifferenceInts that computes the difference between two given sets of intervals. Algorithm 13 computes a possibly inconsistent sequence map. Since the input is either single-variable or single-edge we refrain from defining a consistent version as it can be achieved with the same complexity by a $\dddot{\omega}_v^{\uparrow}$ followed by $\dddot{\omega}_v^{\downarrow}$ where $v$ is either the single variable covered by the two sequence maps or the child variable in the edge cover.

Since we only consider single-edge sequence maps, the disadvantages of the join with overlapping edge sets (in contrast to the one with disjoint edge sets) are not exhibited by the sequence map difference.

**Theorem 12.11.** *Algorithm 14 computes a minimal, non-overlapping sequence of intervals representing the union of two sequences of intervals $I_1$ and $I_2$ in $\mathcal{O}(|I_1| + |I_2|)$ time and constant additional space.*

*Proof.* Algorithm 14 computes minimal, non-overlapping intervals by the same observation as for Algorithm 8: an interval is added only, if it is clear that the next binding after its end index is covered also in the second sequence map and thus should not be included in the result (line 7 and 11).

The result of Algorithm 14 is the difference of the intervals in the input sequences: if a binding is covered by an interval in $I_1$ but not in $I_2$, then it is either entirely before any interval in $I_2$ (lines 6–8) or it is partially overlapping (lines 10–13) in which case we split, if some prefix of it is not covered in $I_2$ (line 11) and continue with any suffix (if there is any).

The algorithm relies on the fact that the interval sets are ordered by increasing start index and non-overlapping. This allows us to infer that, if an interval from $I_1$ lies before the first (remaining) interval from $I_2$, then it lies before *all* intervals of $I_2$ and vice versa. Thus deciding overlapping becomes amortised constant rather than linear in the size of the intervals.

Algorithm 14 runs in $\mathcal{O}(|I_1| + |I_2|)$ as in each iteration of the loop 3–14 either $i$ or $j$ or both are incremented. Getting the next interval from $I_1$ and $I_2$ (line 4–5) is by assumption constant, as is adding an interval at the end of NextIntervals (lines 7, 11). □

**Algorithm 13**: $\ddot{\setminus}(S_1, S_2)$

**input** : Single-variable or single-edge Sequence maps $S_1$ and $S_2$ as in Definition 12.8

**output**: Sequence map res representing the difference of the induced relations of the input maps

```
1  SharedVars ← dom S₁ (= dom S₁ ∩ dom S₂); res ← S₁;
2  if |SharedVars| = 1 then
       // Single-variable sequence maps
3      let v be the single variable in SharedVars;
4      res(v) ← ∅;
5      while i ≤ |S₁(v)| do
           // bindings returns ∞ for index out of bound
6          n₁ ← binding(S₁(v)[i]); n₂ ← binding(S₂(v)[j]);
7          if n₁ = n₂ then i++; j++;
8          else if n₁ < n₂ then
9              res(v)[k] ← (n₁, I);
10             i++; k++;
11         else if n₁ = ↯ then i++;
12         else j++;
13 else
       // Single-edge sequence maps
14     let c be the child, p the parent variable in SharedVars;
15     i, j, k ← 1;
16     while i ≤ |S₁(p)| do
           // bindings returns ∞ for index out of bound
17         n₁ ← binding(S₁(v)[i]); n₂ ← binding(S₂(v)[j]);
           // intervals returns ∅ for index out of bound
18         if n₁ = n₂ then
19             I ← DifferenceInts(π_{2,3}(intervals(S₁(v)[i])),
                                  π_{2,3}(intervals(S₂(v)[j])));
20             if I = ∅ then n₁ ← ↯ else res(v)[k] ← (n₁, I); k++;
21             i++; j++;
22         else if n₁ < n₂ or n₁ = ↯ then
23             res(v)[k] ← (n₁, I);
24             i++; k++;
25         else j++;
26 return res
```

**Algorithm 14**: DifferenceInts($\text{Intervals}_1$, $\text{Intervals}_2$)

**input** : Two sequences $\text{Intervals}_1$, $\text{Intervals}_2$ of non-overlapping intervals (*without* associated variables) in order of start index

**output**: Minimal, non-overlapping sequence of intervals covering all indices contained in intervals of the first input sequence, but not in the second

```
1  NewIntervals ← ∅ ;
2  i, j, k ← 1;
3  while i ≤ |Intervals_1| or j ≤ |Intervals_2| do
4      (v, s_1, e_1) ← Intervals_1[i] or ∞ if i > |Intervals_1|;
5      (v, s_2, e_2) ← Intervals_2[i] or ∞ if j > |Intervals_2|;
6      if e_1 < s_2 then                          // (s_1, e_1) before (s_2, e_2)
7          NewIntervals[k] ← (s_1, e_1); k++;
8          i++;
9      else if e_2 < s_1 then j++ ;               // (s_2, e_2) before (s_1, e_1)
10     else                                       // (s_1, e_1) overlaps (s_2, e_2)
11         if s_1 < s_2 then NewIntervals[k] ← (s_1, s_2 − 1); k++;
12         if e_2 < e_1 then Intervals_1[i] ← (e_2 + 1, e_1);
13         j++;
14         else i++;
15
16 return NewIntervals ;
```

## 12.5 REDUCE

Given a single input map, we can perform several operations on that sequence map in isolation that drop certain variables (projection), certain bindings (selection), or propagate changes from bindings of one variable to those for another (propagate) removing any inconsistencies w.r.t. the propagated pair.

Sequence maps are closed under projection and propagation, but under selection only if we allow only conditions that refer only to one variable or to two variables adjacent in the edge cover of the sequence map.

### 12.5.1 PROJECT

Projection is defined analogous to relational projection: We retain only bindings for variables specified in the projection, bindings for other variables are dropped (including interval pointers to those bindings). Formally, we define the sequence map projection as follows:

**Definition 12.9** (Sequence map projection)**.** Let $D$ be a relational structure, $Q$ a tree query, $S$ a sequence maps for $D$ over $Q$, and $V \subset \operatorname{dom} S$ a sub-set of variables in $Q$ such that for each pair $(v, v') \in V^2$ either $v, v'$ have no least common ancestor, $\operatorname{lca}(v, v')$, in the edge cover of $S$ or all variables on the path from $\operatorname{lca}(v, v')$ to $v$ and $v'$, respectively, are also in $V$.

Then, $\dddot{\pi}_V(S)$ returns a sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that

**(1)** the induced relation $R_{\overset{D,Q}{\mathbf{sm}}_3}$ of $\overset{D,Q}{\mathbf{sm}}_3$ is $R_{\overset{D,Q}{\mathbf{sm}}_3} = \pi_V(R_S)$, i.e., the projection to $V$ of the induced relations of $S$.

**(2)** $\overset{D,Q}{\mathbf{sm}}_3|_{\operatorname{dom} S} = \overset{D,Q}{\mathbf{sm}}_3 (= \overset{D,Q}{\mathbf{sm}}_3|_V)$ (contains bindings only for variables mapped in $S$).

The associated edge cover for $\dddot{\pi}_V(S)$ is $\mathsf{edgeCover}(S) \cap V^2$.

The restriction on the shape of $V$ ensures, that there is no variable with parent in $\operatorname{dom} S \setminus V$ but ancestor in $V$ w.r.t. the edge cover of $S$. In that case, the parent must be removed, however, it is needed to represent which bindings for the ancestor relate to which bindings of the descendant. Simply dropping the parent's bindings yields a cross product between the bindings of ancestor and descendant and thus more tuples than allows by the definition. The alternative to the restriction is that the resulting sequence map is no longer over $D$ and $Q$ but over a query $Q'$ and relational structure $D'$ such that **(1)** the variables of $Q'$ are as in $Q$ and the edges are $\mathsf{Edges}(Q) \setminus (V \times \operatorname{dom} S \cup \operatorname{dom} S \times V) \cup \mathsf{Edges}(Q) \cap V^2 \cup \mathcal{E}$ with $\mathcal{E}\{(v_1, v_k) \in V : \exists v_2, \ldots, v_{k-1} \in \operatorname{dom} S \setminus V : (v_1, v_2), \ldots, (v_{k-1}, v_k) \in \mathsf{Edges}Q\}$. **(2)** For each such edge $(v_1, v_k) \in \mathcal{E}$, there is a new relation name $R$ in the relational schema for $D'$ that is the label of that edge in $Q'$ and relation instance

$R^{D'}$ in $D'$ such that $R^{D'} = \pi_{1,k}(R_1 \bowtie_{2=1} R_2 \bowtie_{2=1} \ldots \bowtie_{2=1} R_{k-1})$ if $R_i$ is the relation name of the edge between $v_i$ and $v_{i+1}$. In the following, we only present the first approach.

The above definition allows both consistent and inconsistent sequence maps as result of $\ddot{\pi}$. However, Algorithm 15 ensures that, if the input sequence map is consistent, the result is consistent. If the input sequence map is inconsistent, the result is only consistent, if the inconsistencies are limited to bindings of variables not contained in the projection set.

Algorithm 15 gives a straightforward implementation for the sequence map projection: For all variables in the projection set $V \subset \operatorname{dom} S$ we either copy the bindings of the variable unchanged, if none of its children (in the edge cover of the input sequence map) are dropped, or copy them retaining only interval pointers to variables in $V$ (line 7 $\sigma_{1 \in V}$ selects those tuples from $\mathsf{intervals}(S(v)[i])$ that have a variable from $V$ as first component).

**Algorithm 15**: $\ddot{\pi}_V(S)$

**input** : Sequence map $S$ over a query $Q$ and set of variables $V \subset \operatorname{dom} S$

**output**: Sequence map res representing the projection of $S$ to the variables in $V$

```
1 res ← ∅; // All variables with a covered edge to a dropped child
2 DropParent ← {v ∈ V : ∃v' ∈ dom S \ V : (v, v') ∈ edgeCover(S)};
3 foreach v ∈ V do
4     if v ∈ DropParent then
5         res(v) ← ∅;
6         foreach i ← 1 to |S(v)| do
7             res(v)[i] ← (binding(S(v)[i]), σ_{1∈V}(intervals(S(v)[i])));
8     else res(v) ← S(v);
9 return res
```

**Theorem 12.12.** *Algorithm 15 computes $\ddot{\pi}_V(S)$ for a sequence map $S$ and a projection set $V \subset \operatorname{dom} S$ in $\mathcal{O}(b^{total}_{DropParent} \cdot i + |V|) \leq \mathcal{O}(q \cdot n \cdot i)$ time and constant additional space where $i$ is the maximal number of intervals per binding and $b^{total}_{DropParent}$ is the total number of bindings for variables with children in $\operatorname{dom} S \setminus V$.*

*Proof.* The resulting sequence map for Algorithm 15 retains bindings (line 3) and interval pointers (line 7) only for variables from $V$. For variables from $V$ neither the bindings nor the interval pointers are touched. Together with the edge cover $\{(v, v') \in \mathsf{edgeCover}(S) : v, v' \in V\}$ as in

Definition 12.9, this yields an induced relation that is the projection to $V$ of the induced relation of the input sequence map.

The Algorithm loops over all variables in $V$ and either copies the bindings unchanged (line 8) or touches each binding and modifies it by dropping the non-$V$ interval pointers (line 4–7), thus touching each (of maximum $i$) intervals associated with that binding.

If the data is tree, forest, or CIG shaped, $i = 1$. □

Note, that each variable has a unique parent, if any, in a tree query. Thus, even if a CIQCAG expression for evaluating a given tree query drops all but one variable, whether in a single project operation, or in a sequence of $q - 1$ projections, at most $q - 1$ variables become "drop parents", i.e., variables with dropped child variable, over the whole expression and, thus, the overall complexity for the projections is also bounded by $\mathcal{O}(q \cdot n \cdot i)$.

Obviously, the result of a projection is a sequence map and $\ddot{\pi}_V$ can be applied to any sequence map $S$ with $V \subset \operatorname{dom} S$.

### 12.5.2 SELECT

Sequence map selection we also based on relational selection: We retain only bindings that fulfill a given selection condition. However, in contrast to relational selection, conditions may not relate multiple variables. Rather each condition may only reference a single variable from the query. This restriction is necessary to ensure the tree query property, where only relations between variables adjacent in the tree query are allowed. Consider, e.g., the sequence map $S$ shown in Figure 63: A selection on the induced relation of $S$ with a condition $v_1 = v_2$ (reading = as node identity) yields the relation $\{(d_1, d_3, d_1), (d_2, d_3, d_2)\}$. This relation can not be represented as a sequence map, since it can no longer be decomposed into binary relations over the pairs of adjacent variables without loss: only the $d_1$ (not the $d_2$) binding for $v_1$ remains related to the $d_1$ binding for $v_2$. However, both remain related to $d_3$ (as $(d_2, d_3, d_2)$ remains in the relation). Thus, either we allow also $(d_2, d_3, d_1)$ (and $(d_1, d_3, d_2)$ by retaining an interval pointer 1–2 in $d_3$ or we drop all references from $d_3$ to bindings for $v_3$, thus yielding a sequence map representing the empty relation.

Thus we define the sequence map selection as follows:

**Definition 12.10** (Sequence map selection)**.** Let $D$ be a relational structure, $Q$ a tree query, $S$ a sequence maps for $D$ over $Q$, and $c$ a single atom containing (possibly multiple) references to a *single variable*. Then $\ddot{\sigma}^{\not{\iota}}_c(S)$ returns a sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that

**(1)** the induced relation $R_{\overset{D,Q}{\mathbf{sm}}_3}$ of $\overset{D,Q}{\mathbf{sm}}_3$ is $R_{\overset{D,Q}{\mathbf{sm}}_3} = \sigma_c(R_S)$, i.e., the selection on $c$ over the induced relations of $S$.

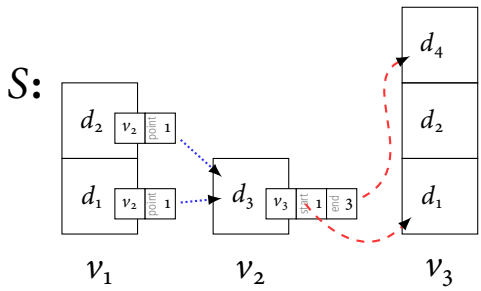



Figure 63. Sequence maps for illustrating the single-variable condition restriction for $\ddot{\sigma}$

**(2)** $\overset{D,Q}{\mathbf{sm}}_3|_{\mathrm{dom}\,S} = \overset{D,Q}{\mathbf{sm}}_3$ (contains bindings only for variables mapped in $S$). The associated edge cover for $\ddot{\sigma}^{\lightning}_V(S)$ is the same as for $S$.

We allow in the definition only a single atom, since conjunctions, disjunctions, and quantification can be achieved by combining selection with union, join, or difference. As a convenience, we can easily extend $c$ to conjunctions and/or disjunctions of atoms, all over the same single variable, without changes to the algorithms and results.

Algorithm 16 gives a straightforward implementation for the inconsistent case: Here, we merely "bomb" all bindings where $c$ does not hold if we replace $v$ with its binding in $c$ (line 3). Since the number and order of bindings is untouched, the interval pointers can remain unchanged (but now possibly point to some or all failure markers).

**Algorithm 16**: $\ddot{\sigma}^{\lightning}_c(S)$

**input** : Sequence map $S$ over a query $Q$ and condition $c$ on variable $v \in \mathrm{dom}\,S$

**output**: Possibly inconsistent sequence map res representing the selection of only bindings for $v$ from $S$ matching $c$

```
1 res ← S;
2 foreach i ← 1 to |res(v)| do
3     if ¬c{v/res(v)[i]} then res(v)[i] ← ↯;
4 return res
```

**Theorem 12.13.** *Algorithm 16 computes $\ddot{\sigma}^{\lightning}_V(S)$ for a sequence map $S$ and a condition $c$ over a single variable $v$ in $\mathcal{O}(b_v \cdot m_c) \leq \mathcal{O}(n \cdot m_c)$ time and constant additional space where $m_c$ is the maximum time for evaluating $c$*

*if all references to $v$ are replaced by a binding for $v$ in $S$ and $b_v$ the number of bindings for $v$ in $S$. For most conditions, $m_c$ is constant.*

*Proof.* Algorithm 16 computes $\ddot{\sigma}^{\not{f}}_V(S)$: No $v$ binding for that $c$ does not hold is retained (line 2–3). All bindings for other variables are untouched (line 1).

It is obviously bounded by the number of bindings for $v$ in $S$ (line 2), for each of which $c$ is evaluated in line 3. □

*Consistent sequence map selection*

Definition 12.10 allows the input to be inconsistent and a inconsistent sequence maps as result of $\ddot{\sigma}^{\not{f}}$. As for most other operators, we can drop this restrictions and require consistent sequence maps as input and result. We denote the resulting operation with $\ddot{\sigma}$. The required changes are similar to the case of the sequence map join (cf. Section 12.4.1): We introduce a change log for the changes applied to the variable $v$ involved in $c$ and use this change log to adapt the intervals of the parent variable of $v$. Since we may also drop bindings for the parent variable (where all related bindings for $v$ are dropped), we need to propagate such changes upwards in the query to all ancestors to $v$. The corresponding algorithm is given in Algorithm 17 and shows roughly the same relation to the above Algorithm for $\ddot{\sigma}^{\not{f}}$ as Algorithm 7 to Algorithm 5.

**Theorem 12.14.** *Algorithm 17 computes $\ddot{\sigma}_V(S)$ for a consistent sequence map $S$ and a condition $c$ over a single variable $v$ in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot (\max(i, m_c)) \leq \mathcal{O}(q \cdot n \cdot \max(i, m_c))$ time and $\mathcal{O}(n)$ additional space where $m_c$ is the maximum time for evaluating $c$ if all references to $v$ are replaced by a binding for $v$ in $S$, $i$ the maximum number of associated intervals for a binding in $S$, and $b_{\mathcal{V}}^{total}$ the total number of bindings for all variables in $\mathcal{V} = \{v' \in \operatorname{dom} S : \exists$ path from $v'$ to $v$ in edgeCover$(S)\}$.*

### 12.5.3 PROPAGATE

As discussed above, in particular for the sequence map join in Section 12.4.1, many of the previous operators are easier to implement with better complexity if we allow the results to be inconsistent. Though this advantage may, to some extent, be offset by the increase in result size, if the involved operations are selective, we can still profit in many cases from inconsistent sequence maps: Instead of propagating changes from a often small subset of variables involved in an operation to all variables after each operation, we propagate these changes once at the end of an entire sequence of operations.

For some operations, however, a *consistent* sequence map is either required or beneficial. This is particularly true for $\mathit{F}$, which extracts bindings

**Algorithm 17**: $\ddot{\sigma}_c(S)$

**input** : Consistent sequence map $S$ over a query $Q$ and condition $c$ on variable $v \in \operatorname{dom} S$

**output**: Consistent sequence map res representing the selection of only bindings for $v$ from $S$ for which $c$ holds

```
1  EC ← edgeCover(S);
2  res ← S;
   // Process the selection variable
3  res(v) ← ∅; k ← 1; Log ← ∅; last ← ∞;
4  for i ← 1 to |S(v)| do
5      if c{v/S(v)[i]} then
6          res(v)[k] ← S(v)[i];
7          Log ← Log ∪ {(v, ↓, i, k)} ∪ {(v, ↪, l, i) : last ≤ l < i};
8          k++; last ← ∞;
9      else
10         last ← min(last, i); Log ← Log ∪ {(v, ↩, i, last − 1)};
11 if last ≠ ∞ then
12     Log ← Log ∪ {(v, ↪, l, ∞) : last ≤ l ≤ |S(v)|}
   // Process the ancestors of the selection variable
13 Ancestors ← {v′ ∈ dom S : ∃ path from v′ to v in EC}
14 foreach v′ ∈ Ancestors in topological order w.r.t. EC do
15     NoDrop ← true;
16     res(v′) ← ∅; k ← 1; last ← ∞;
17     for i ← 1 to |S(v′)| do
18         I ← Adapt(intervals(S(v′)[i]), Log);
19         if I = ∅ then
20             last ← min(last, i);
21             Log ← Log ∪ {(v′, ↩, i, last − 1)};
22             NoDrop ← false;
23         else
24             res(v′)[i] ← (binding(S(v′)[i]), I);
25             Log ← Log ∪ {(v′, ↓, i, k)} ∪ {(v, ↪, l, i) : last ≤ l < i};
26             k++; last ← ∞;
27     if NoDrop then break;
28     if last ≠ ∞ then
29         Log ← Log ∪ {(v′, ↪, l, ∞) : last ≤ l ≤ |S(v)|}
30 return res
```

for some variables from a sequence map. Consider, e.g., that we are interested only in the bindings of a single variable $v$. If the given sequence $S$ map is consistent, it suffices to return the bindings in $S(v)$, without even considering the other variables or the relation of them to $v$. This is possible, since every binding $n \in S(v)$ occurs in at least one tuple of the induced relation of $S$ (i.e., can be extended to a full answer if $S$ is complete). If, however, $S$ may be inconsistent, a binding $n \in S(v)$ (or, for that matter, its parents or their parents …) may be dangling in which case there is no binding for $v$'s parent and $n$ does *not* contribute to a tuple in the induced relation. If $n$ is a failure marker, it does not contribute to the induced relation by definition and, worse, bindings for $v$'s parent may only be related to failure markers among the binding for $v$ and thus, in fact, also not contribute to the induced relation.

To enable this mode of processing, we require explicit propagation operators, that, progressively, ensure that a resulting sequence map is consistent. For the two types of inconsistency, two different operators are provided:

**(1)** $\dddot{\omega}^{\uparrow}_{v}(S)$ removes all failure markers from the bindings of $v$ and, if there is a parent $v'$ of $v$ covered by $S$, adopts the interval pointers of $v$ accordingly. This may lead to some bindings of $v'$ without related bindings for $v$. These bindings can no longer contribute to a tuple in the induced relation of $S$ and are thus "bombed". This way we move the consistency up one level in the (tree) query, from $v$ to its unique parent $v'$. Since $v'$ is the only variable whose bindings have interval pointers to bindings of $v$, no other variable is directly affected. However, to remove all failure markers from $S$, we need to perform $\dddot{\omega}^{\uparrow}_{v}(S)$ for all variables covered by $S$ in inverse topological order (i.e., child before parent).

**(2)** $\dddot{\omega}^{v}_{\downarrow}(S)$ discovers and removes all "dangling" bindings of $v$. Note, that this may affect the interval pointers from $v' = \mathsf{parent}(v)$ to $v$: They may "slide" to the left, as "dangling" bindings for $v$ are removed from the beginning of the binding sequence $S(v)$, and merge with the previous interval(s), if all the intermediary bindings are dangling. The number of related bindings of $v$, however, remains unchanged for each binding of $v'$. Therefore, the sequence of bindings for $v'$ remains the same, only the interval pointers change. To remove all "dangling" bindings from $S$, we need to perform $\dddot{\omega}^{v}_{\downarrow}$ for all non-root variables covered by $S$ in topological order (i.e., parent before child).

If we perform first $\dddot{\omega}^{\uparrow}_{v}(S)$ for all variables by $S$ in inverse topological order, then $\dddot{\omega}^{v}_{\downarrow}(S)$ in the inverse order, the resulting sequence map is consistent, see Theorem 12.17.

Formally, we define the up-propagation operator as follows:

**Definition 12.11** (Sequence map propagation)**.** Let $D$ be a relational structure, $Q$ a tree query, $S$ a sequence maps for $D$ over $Q$, and $v \in \operatorname{dom} S$ a variable covered by $S$. Then $\ddot{\omega}^{\uparrow}_{v}(S)$ ($\ddot{\omega}^{v}_{\downarrow}(S)$) returns a sequence map $\overset{D,Q}{\mathbf{sm}}_3$ such that:

**(1)** the induced relation $R_{\overset{D,Q}{\mathbf{sm}}_3}$ of $\overset{D,Q}{\mathbf{sm}}_3$ is unchanged, i.e., $R_{\overset{D,Q}{\mathbf{sm}}_3} = R_S$.

**(2)** $\overset{D,Q}{\mathbf{sm}}_3|_{\operatorname{dom} S} = \overset{D,Q}{\mathbf{sm}}_3$ (contains bindings only for variables mapped in $S$).

For $\ddot{\omega}^{v}_{\downarrow}(S)$ in addition to 1 and 2 it also holds that

**(3a)** there are no failure markers in $\overset{D,Q}{\mathbf{sm}}_3(v)$ (among the bindings for $v$).

For $\ddot{\omega}^{v}_{\downarrow}(S)$ in addition to 1 and 2 it also holds that

**(3b)** there are no direct dangling bindings in $\overset{D,Q}{\mathbf{sm}}_3(v)$, i.e., no bindings of $v$ that are not covered by some interval pointer of a binding for $v$'s parent in the edge cover of $S$, if there is such a parent.

The associated edge cover for $\ddot{\omega}^{\uparrow}_{v}(S)$ ($\ddot{\omega}^{v}_{\downarrow}(S)$) is the edge cover of the input sequence map.

Note that, for of $\ddot{\omega}^{v}_{\downarrow}(S)$, if $v$ is a root node in $Q$ or $v' = \mathsf{parent}(v)$ in $Q$ but $(v', v) \notin \mathsf{edgeCover}(S)$, the above condition (3b) implies that no binding of $v$ is dangling.

We require that w.r.t. $v$ (and only $v$) the result of $\ddot{\omega}^{\uparrow}_{v}(S)$ ($\ddot{\omega}^{v}_{\downarrow}(S)$) is consistent wrt. failure markers (direct dangling bindings) even if $S$ contains failure markers (direct dangling bindings) for $v$. However, for all other variables there may still be (direct) dangling bindings or failure markers. In fact, to remove failure markers from the bindings of $v$ we need, in general, to adapt the bindings of any variable with an edge to $v$ in the edge cover. Since $Q$ is a tree query, there is a single such variable, $v'$. When we adapt the interval pointers referring from bindings of $v'$ to bindings of $v$, we may end up with bindings for $v'$ with no corresponding binding for $v$. Those bindings are then "bombed". Thus, we may actually *introduce* failure markers for $v'$. Those failure markers may, e.g., be propagated and removed by a later $\ddot{\omega}^{\uparrow}_{v'}(S')$ where $S'$ is the result of $\ddot{\omega}^{\uparrow}_{v}(S)$.

Algorithm 18 shows in detail, how to compute $\ddot{\omega}^{\uparrow}$. It is, basically, the "consistency" component of Algorithm 7 for a single variable. It is divided in two phases, one for $v$ and one for its parent, if there is any: **(1)** In lines 2–9, we remove all failure markers from $S(v)$ and store the result in $S'$. This yields a change log mapping indices for bindings in $S(v)$ to indices in $S'$. The change log is shaped exactly the same as the change log for Algorithm 7: For each retained binding a $\downarrow$ entry associating the old index to the index in $S'$. For each dropped binding (i.e., failure marker), a $\hookleftarrow$ and a $\hookrightarrow$ entry referencing the index of the last preceding and next following retained binding. **(2)** This change log is used in the second phase, lines 10–16, to adapt each of the interval pointers from bindings for $v'$ to bindings for $v$. At the end of phase 2 all intervals reference now indices in $S'$ rather than in

$S(v)$. Note, that, if all interval pointers of a binding are dropped (because they pointed only to failure markers), that binding is itself "bomb"ed (line 13).

**Algorithm 18:** $\ddot{\omega}^{\uparrow}_{v}(S)$

**input** : Sequence map $S$ over a query $Q$ and variable $v \in \operatorname{dom} S$
**output**: Sequence map res representing the induced relation for $S$ with no failure markers among the bindings of $v$

```
1  EC ← edgeCover(S); res ← S;
   // 1—Child Phase: For v, remove failure markers
2  S' ← ∅; k ← 1; Log ← ∅; last ← ∞;
3  for i ← 1 to |S(v)| do
4      if S(v) = ↯ then last ← min(last, i);
       Log ← Log ∪ {(v, ↩, i, last − 1)};
5      else
6          S'[k] ← S(v)[i];
7          Log ← Log ∪ {(v, ↓, i, k)} ∪ {(v, ↪, l, i) : last ≤ l < i};
8          k++; last ← ∞;
9  if last ≠ ∞ then Log ← Log ∪ {(v, ↪, l, ∞) : last ≤ l ≤ |S(v)|}
   // 2—Parent Phase: For parent v', shrink intervals
10 if (v', v) ∈ EC for a v' ∈ dom S then
11     take that v' (there is at most one);
12     res(v') ← ∅;
13     for i ← 1 to |S(v')| do
14         I ← Adapt(intervals(S(v')[i]), Log);
15         if I = ∅ then res(v')[i] ← ↯;        // We introduce a failure
           marker!
16         else res(v')[i] ← (binding(S(v')[i]), I);
17 return res
```

**Theorem 12.15.** *Algorithm 18 computes $\ddot{\omega}^{\uparrow}_{v}(S)$ for a sequence map $S$ and variable $v \in \operatorname{dom} S$ $\mathcal{O}(b_v + b_{v'} \cdot i) \leq \mathcal{O}(n \cdot i)$ time and $\mathcal{O}(b_v) \leq \mathcal{O}(n)$ additional space where $i$ is the maximum number of associated intervals pointing to $v$ for any binding in $S$, and $b_v$ ($b_{v'}$) the number of bindings for $v$ ($v'$ if $v' \in \operatorname{dom} S$ with $(v', v) \in \mathit{edgeCover}(S)$ or 0 otherwise) in $S$.*

*Proof.* Algorithm 18 computes a sequence map $S'$ such that $S'(v)$ contains no failure markers. It contains no failure markers due to line 4.

$S'$ has the same induced relation as the input sequence map, as only

failure markers are dropped which do not contribute to a tuple contained in the induced relation by Definition 11.5.

Phase 1 of the algorithm runs in $\mathcal{O}(b_v)$ time as it iterates over all bindings for $v$ in $S$. For each such binding, the entry is copied to the new sequence map and the change log is updated (in constant time).

Phase 2 runs in constant time if there is no parent in EC, otherwise in $\mathcal{O}(b_{v'} \cdot i)$ time, as it iterate over the bindings for $v'$ in $S$. For each such binding, they call Adapt which runes, by Theorem 12.5 in time $i$ where $i$ is the maximum number of intervals pointing to $v$ associated with a binding of $v'$ (which are the only bindings pointing to $v$). □

For $\ddot{\omega}_{\downarrow}$, we obtain a similar algorithm 19 with three phases: In the first phase, the *skip index* for the parent $v'$ of $v$ (if there is any) is created: It contains for each index $i$ of a binding of $v$ the maximum end index of any interval pointer starting at $i$ and associated with a binding of $v'$. The second phase uses the skip index find all dangling bindings in a single pass over the bindings of $v$. In a third phase, the interval pointers from $v'$ to $v$ are slided (and, possibly, merged) to adapt to the new sequence of bindings for $v$. In detail: **(1)** In the first parent phase, we build a skip index Skip that initially points to 0 for all indices in $S'$. Whenever we find an (adapted, i.e., already pointing to $S'$) interval that starts at an index $s$, we set the skip index for $s$ to the maximum of its current value and the end index $e$ of the interval.

At the end of phase 2 the skip index records, for each index $i$, the maximum end index of any interval that starts at $i$ (or 0 if there is none). **(2)** The skip index is used in the second phase, lines 17–25, for a second pass over the bindings of the child variable. Here, we consider for each binding, if it lies within a interval of one of the bindings for the parent variable. This is recognized using the skip index: maxEnd always points to the highest end point of any interval whose start index we have already passed. If maxEnd is smaller than the current index and the current index is not a start index ($\mathsf{Skip}(i) = 0$), we have found a dangling binding that must be eliminated. Otherwise, we copy the binding and update maxEnd (with the maximum end index of an interval that starts at the current index, if it is larger than maxEnd). **(3)** Finally, in a third phase we slide the intervals of all bindings for the parent variable to adapt to the changes to the sequence of bindings for the child variable in the previous step. Note, that this step does not introduce any new dangling bindings, as intervals do not shrink in this phase, but only slide since the bindings associated with start and end indices can not have been dropped in the previous phase (they are, after all, covered by at least the current interval). Thus, there are $\downarrow$ entries in the change log for them (line 24) and the interval

is simply slided according to those entries and possibly collapsed with a previous interval (if the only bindings between the two intervals have been dangling bindings and thus dropped in the previous phase.

**Algorithm 19**: $\ddot{\bar{\omega}}^{v}_{\downarrow}(S)$

**input** : Sequence map $S$ over a query $Q$ and variable $v \in \operatorname{dom} S$
**output**: Sequence map res representing the induced relation for $S$ with no direct dangling bindings among the bindings of $v$

```
1  EC ← edgeCover(S); res ← S;
2  if (v', v) ∈ EC then
3      take that unique v';
       // 1—Parent Phase: Compute skip index
4      Skip ← {(i, 0) : 1 ≤ i ≤ |S'|}; res(v') ← ∅;
5      foreach (n, I) ∈ S(v') do
6          foreach (s, e) ∈ I do Skip(s) ← max(Skip(s), e);
       // 2—Child Phase: For v, drop dangling bindings
       // maxEnd maximum end point of any "open" interval
7      maxEnd ← 0; res(v) ← ∅; k ← 1; Log ← ∅; last ← ∞; i ← 1;
8      for i ← 1 to |S'| do
9          if maxEnd < i and Skip(i) = 0 then
10             last ← min(last, i); maxEnd ← 0;
11         else
12             res(v)[k] ← S(v)[i];
13             Log ← Log ∪ {(v, ↓, i, k)};
14             k++; last ← ∞; maxEnd ← max(maxEnd, Skip(i));
       // 3—Parent Phase: For v', slide intervals
15     for i ← 1 to |res(v')| do
16         if res(v') = ↯ then continue;
17         I ← Adapt(intervals(res(v')[i]), Log);
18         res(v')[i] ← (binding(res(v')[i]), I);
19 return res
```

**Theorem 12.16.** *Algorithm 19 computes $\ddot{\bar{\omega}}^{v}_{\downarrow}(S)$ for a sequence map $S$ and variable $v \in \operatorname{dom} S$ $\mathcal{O}(b_v + b_{v'} \cdot i) \leq \mathcal{O}(n \cdot i)$ time and $\mathcal{O}(b_v) \leq \mathcal{O}(n)$ additional space where $i$ is the maximum number of associated intervals pointing to $v$ for any binding in $S$, and $b_v$ ($b_{v'}$) the number of bindings for $v$ ($v'$ if $v' \in \operatorname{dom} S$ with $(v', v) \in \mathit{edgeCover}(S)$ or 0 otherwise) in $S$.*

*Proof.* Algorithm 19 computes a sequence map $S'$ such that $S'(v)$ contains

no direct dangling bindings. Due to lines 17–25, it contains no direct dangling binding, since, for each direct dangling binding $n$ with index $i$, $\mathsf{Skip}(i) = o$ (there can be no interval with $i$ as start index otherwise $n$ is not dangling) and there is no previous index $j \leq i$ that is the start index of an interval that cover $i$ (i.e., with end index $\geq i$). Thus $\mathsf{maxEnd} < i$, all conditions of line 19 are fulfilled, and the binding is dropped.

$S'$ has the same induced relation as the input sequence map, as only dangling bindings are dropped which do not contribute to a tuple contained in the induced relation by Definition 11.5.

Phase 1 and 3 of the algorithm run in $\mathcal{O}(b_{v'} \cdot i)$ time, as they both iterate over the bindings for $v'$ in $S$. For each such binding, phase 1 iterates over all its $i$ intervals and updates the skip index (single comparison and assignment). Phase 3 calls Adapt which runs, by Theorem 12.5, in time $i$ where $i$ is the maximum number of intervals pointing to $v$ associated with a binding of $v'$.

Phase 2 of the algorithm runs in $\mathcal{O}(b_v)$ time as they both iterate at most over all bindings for $v$ in $S$ (phase 3 already skips failure markers). For each such binding, the entry is copied to the new sequence map and the change log is updated (in constant time). □

From these results, we can immediately conclude the following result:

**Theorem 12.17.** *Let* $\overset{D,Q}{\mathsf{sm}}$ *be an inconsistent sequence map. Then there is a consistent sequence map* $\overset{D,Q}{\mathsf{sm}}'$ *equivalent to* $\overset{D,Q}{\mathsf{sm}}$*. This sequence map can be computed in* $\mathcal{O}(\tilde{q} \cdot n \cdot i)$ *where* $\tilde{q} = |\mathrm{dom}\,\overset{D,Q}{\mathsf{sm}}|$, $n = |\mathit{Nodes}(D)|$, *and* $i$ *the maximum number of intervals per binding in* $S$*. For tree, forest, and* CIG *data* $i = 1$.

*Proof.* To compute the consistent sequence map $S'$ from the inconsistent input sequence map $S$, we use a ClQCAG-expression using $\ddot{\bar{\omega}}^{\uparrow}$ and $\ddot{\bar{\omega}}_{\downarrow}$. Let $\mathrm{dom}\, S = \{v_1, \ldots, v_k\}$ such that, for any $v_i, v_j$ with $v_i < v_j$ wrt. the topological order on $\mathrm{dom}\, S$ induced by $\mathsf{edgeCover}(S)$ (i.e., $v < v'$ if $\exists$ path from $v'$ to $v$ in $\mathsf{edgeCover}(S)$), it holds that $i < j$. Then the following ClQCAG-expression computes a consistent sequence map that is equivalent to $S$:

$$\ddot{\bar{\omega}}^{v_1}_{\downarrow}(\ddot{\bar{\omega}}^{v_2}_{\downarrow}(\ldots\ddot{\bar{\omega}}^{v_k}_{\downarrow}(\ddot{\bar{\omega}}^{\uparrow}_{v_k}(\ddot{\bar{\omega}}^{\uparrow}_{v_{k-1}}(\ldots(\ddot{\bar{\omega}}^{\uparrow}_{v_1}(S))\ldots)))\ldots))$$

The resulting sequence map is consistent, as it contains: **(1)** no failure markers for any $v_i$, as there is a $\ddot{\bar{\omega}}^{\uparrow}_{v_i}(\mathcal{E}_i)$ for each $v_i$ and only $\ddot{\bar{\omega}}^{\uparrow}_{v_j}(\mathcal{E}_j)$ with $j < i$ may create failure markers for $v_i$ (viz. $\ddot{\bar{\omega}}^{\uparrow}_{v}(\mathcal{E}_v)$ for all children $v$ of $v_i$), all of which are contained in $\mathcal{E}_i$. **(2)** no direct dangling bindings for any $v_i$, as there is a $\ddot{\bar{\omega}}^{v_i}_{\downarrow}(\mathcal{E}_i)$ only $\ddot{\bar{\omega}}^{\uparrow}$'s and $\ddot{\bar{\omega}}^{v_j}_{\downarrow}(\mathcal{E}_j)$ with $j > i$ influence the bindings of the parent of $v_i$ and all these are contained in $\mathcal{E}$. **(3)** no indirect dangling bindings as there are no direct dangling bindings.

The CIQCAG-expression has a size of $2 \cdot |\text{dom } S|$ and consists only of $\ddot{\omega}_{\downarrow}$ and $\ddot{\omega}^{\uparrow}$ expressions, all of which operate on $S$ or a sequence map smaller than $S$. Thus, all of them are bound by $\mathcal{O}(n \cdot i)$ where $i$ is the maximum number of intervals per binding in $S$.

Overall, the computation is thus bound by $\mathcal{O}(|\text{dom } S| \cdot n \cdot i)$. □

In the rest of this work, we use often sequences $v_1, \ldots, v_n$ of variables instead of a single variable as index for $\ddot{\omega}^{\uparrow}$ ($\ddot{\omega}_{\downarrow}$). This notation is a shorthand for a sequence of $n$ nested $\ddot{\omega}^{\uparrow}$ ($\ddot{\omega}_{\downarrow}$) expressions each for one variable in the order of the variables in the index sequence.

## 12.6 RENAME

For convenience, we briefly discuss an analog to the rename operator on (named) relational algebra, though it is not used in Chapter 13.

**Definition 12.12** (Sequence map renaming)**.** Let $D$ be a relational structure, $Q$ a tree query, $S$ a sequence maps for $D$ over $Q$, and $v_1 \in \text{dom } S, v_2 \in \text{Vars}(Q) \setminus \text{dom } S$. Then $\ddot{\rho}_{v_1 \to v_2}(S)$ returns a sequence map $\overset{D,Q}{\mathbf{sm}}{}'$ such that

**(1)** the induced relation $R_{\overset{D,Q}{\mathbf{sm}}{}'}$ of $\overset{D,Q}{\mathbf{sm}}{}'$ is $R_{\overset{D,Q}{\mathbf{sm}}{}'} = \rho_{v_1 \to v_2}(R_S)$, i.e., the induced relation of the input sequence map with attribute $v_1$ renamed to $v_2$.

**(2)** $\overset{D,Q}{\mathbf{sm}}{}'|_{\text{dom } S} = \overset{D,Q}{\mathbf{sm}}{}'$ (contains bindings only for variables mapped in $S$).

The associated edge cover for $\ddot{\rho}_{v_1 \to v_2}(S)$ is the edge cover of the input sequence map with all occurrences for $v_1$ replaced by $v_2$.

Note, that $\ddot{\rho}_{v_1 \to v_2}$ can be applied only, if $v_2$ is not covered by $S$ and if $v_2$ has at least the same in- and outgoing edges in $Q$ as $v_1$ has in the edge cover of $S$. This limits the applicability compared to the relational case.

Algorithm 20 shows how to compute the sequence map rename operation. Note, that only bindings for the parent of $v_1$, if there is any in the edge cover of $S$, are processed.

**Theorem 12.18.** *Algorithm 20 computes $\ddot{\rho}_{v_1 \to v_2}(S)$ for a sequence map $S$ and variables and $v_1 \in \text{dom } S, v_2 \in \text{Vars}(Q) \setminus \text{dom } S$ in $\mathcal{O}(b_{\text{parent}(v_1)} \cdot i) \leq \mathcal{O}(n \cdot i)$ time and constant additional space where $b_{\text{parent}(v_1)}$ is the number of bindings for the parent of $v_1$ in $S$ and $i$ the maximum number of associated intervals associated with such a binding.*

**Algorithm 20**: $\dddot{\rho}_{v_1 \to v_2}(S)$

**input** : Sequence map $S$ over a query $Q$ two variables $v_1 \in \operatorname{dom} S$, $v_2 \in \mathsf{Vars}(Q) \setminus \operatorname{dom} S$

**output**: Sequence map res representing the induced relation of $S$ with $v_1$ renamed to $v_2$

```
1  res ← S;
   // All variables with a covered edge to v_1
2  ParentVars ← {v ∈ dom S : (v, v_1) ∈ edgeCover(S)};
3  foreach v ∈ ParentVars do
4      res(v) ← ∅;
5      foreach i ← 1 to |S(v)| do
6          I ← σ_{1≠v_1}(intervals(S(v)[i])) ∪ {v_2} ×
           π_{2,3}(σ_{1=v_1}(intervals(S(v)[i])));
7          res(v)[i] ← (binding(S(v)[i]), I);
8  res(v_2) ← res(v_1);
9  res ← res ∖ {(v_1, res(v_1))};
10 return res
```

## 12.7 BACK TO RELATIONS: EXTRACT

Fittingly, we conclude the introduction and definition of the sequence map operators in CIQCAG with the sequence map extraction, where we obtain the variable bindings for a subset $V$ of a sequence maps variables as a relation. A tuple $t = \langle v_1 : n_1, \ldots, v_k : n_k \rangle$ is contained in that relation, if $V = \{v_1, \ldots, v_k\}$ and $t$ is in the projection to $V$ of the induced relation of the sequence map $S$. Thus, for each $v_i : n_i$ there is an index $l$ such that $\mathsf{binding}(S(v_i)[l]) = n_i$ and, if $(v, v') \in \mathsf{edgeCover}(I)$, then $v : n \in t$ and $v' : n' \in t$ if there are indices $l, l'$ such that $S(v)[l] = (n, I)$, $\mathsf{binding}(S(v')[l']) = n'$ and $l'$ is covered by some interval in $I$. If the data is tree, forest, or CIG shaped, $|I| \leq 1$ and $l'$ must lie within the boundaries of the single interval in $I$.

Formally, we define the sequence map extract operator as follows:

**Definition 12.13** (Sequence map extraction)**.** Let $D$ be a relational structure, $Q$ a tree query, $S$ a sequence maps for $D$ over $Q$ with the induced relation $R_S$, and $V = \{v_1, \ldots, v_k\} \subset \operatorname{dom} S$. Then $\digamma_V(S)$ returns a relation $R = \pi_V(R_S)$.

By definition, $\digamma_{\operatorname{dom} S} S$ returns exactly the induced relation of $S$, yielding an algorithm for computing the induced relation.

**Algorithm 21**: $f_V(S)$

**input** : Sequence map $S$ and variables $V = \{v_1, \ldots, v_k\} \subset \operatorname{dom} S$
**output**: $\pi_{v_1,\ldots,v_k}(R_S)$

```
1 res ← {};
2 RootVars ← {v ∈ dom S : ∄ v' : (v, v') ∈ edgeCover(S)};
3 foreach v ∈ RootVars do
4     res ← res × Relation(S, v, 1, |S(v)|);
5 return π_{v_1,...,v_k}(res) or ∅ if res = {⟨⟩}
```

First, we present a naive version of $f$ that *always* computes the induced relation and simply applies a relational projection to the computed relation. It is presented in Algorithm 21 and uses Algorithm 22 to compute the induced relation for each connected component.

**Theorem 12.19.** *Algorithm 21 computes $f_V(S)$ for a sequence map $S$ and a set of variables $V \subset \operatorname{dom} S$ in $\mathcal{O}(b_+^{|\operatorname{dom} S|}) \leq \mathcal{O}(n^q)$ where $b_+$ is the maximum number of "good" bindings (neither failure markers nor dangling) for a variable in $S$.*

*Proof.* The result of Algorithm 21 is the projection to $V$ of the induced relation of $S$: Relation computes the induced relation for each connected component of the edge cover EC of $S$ rooted at one of the root vars. This is extended to the full induced relation in line 4 (note that there are no covered edges between variables from different connected components and thus a mere cross product suffices.

Algorithm 22 computes the induced relation for each connected component of the edge cover: it combines each binding for the root variable (line 5) with all bindings generated for each of its children (w.r.t. the edge cover of $S$), line 6–9. If all bindings for $v$ are either failure markers or dangling, $\varnothing$ is returned (and, if this is the only all to Relation for $v$, the entire induced relation becomes $\varnothing$.

Algorithm 22 performs, for each root variable (lines 3–4 in Algorithm 21), the expansion of the induced relation over all descendant variables. The outer loop (lines 2–6) iterates over all elements of the given interval. There are at most $b^{int} \leq b \leq n$ such elements. For each such element, it iterates (lines 4–5) over all its associated intervals of which there are at most $i \leq n$ per child variable of which there are at most $b_{\text{EC}}$ where $b_{\text{EC}}$ is the maximum degree of a variable in the edge cover associated with $S$. For each such interval, a recursive call to Relation with an interval of maximum size $b^{int} \leq b \leq n$ (line 5). The recursive calls return, if a leaf variable in the edge cover of $S$ is found, at a recursion depth of at most $d_{\text{EC}}$ where

$d_{\text{EC}}$ is the maximum depth of the edge cover associated with $S$. Overall, Algorithm 22 runs in $\mathcal{O}((b_{\text{EC}} \cdot b^{int} \cdot i)^{d_{\text{EC}}} + o)$ where $o$ is the size of the result relation.

The result relation is bound by $b_+(\tilde{q})^{|\tilde{q}|}$ where $\tilde{q}$ is the number of descendant variables of a node $v$ in EC and $b_+(\tilde{q})$ is the maximum number of "good" bindings for any variable in $\tilde{q}$.

In Algorithm 21, we always compute the full induced relation (lines 3–4) only to drop bindings for variables not in $V$ in line 5. The complexity of Algorithm 21 is dominated by the time and space for construction and storing the induced relation and thus by $\mathcal{O}(b_+^{|\text{dom } S|})$. □

**Algorithm 22**: Relation($S, v$, start, end)

**input** : Sequence map $S$, variable $v \in \text{dom } S$, start and end index
**output**: Relation containing one tuple for each combination of bindings for $v$ and each of its children represented in $S$

```
1  ChildVars ← {v' ∈ dom S : (v, v') ∈ edgeCover(S)}; res ← {⟨⟩};
2  for i ← start to min(end, |S(v)|) do
3      (n, I) ← S(v)[i];
4      if n = ↯ then continue;
5      R ← {⟨v : n⟩};
6      foreach v' ∈ ChildVars ∩ π_1(I) do
7          R' ← R; R ← ∅;
8          foreach (v', start', end') ∈ I do
9              R ← R ∪ (R' × Relation(S, v', start', end'));
10     res ← res ∪ R;
11 return res
```

In most cases, we are not interested to compute the full induced relation, but only in a few of the variables covered by a sequence map. It is also not sufficient to use $\ddot{\pi}$ to remove all the variables we are not interested in as $\ddot{\pi}$ is limited w.r.t. the shape of the variable sets allowed (no variable with parent outside the projection set but ancestor within is allowed). The above algorithm, however, always computes the entire induced relation even for parts of the query not relevant for the variables we are actually interested in and that are specified in $V$.

Therefore, we present a second algorithm Algorithm 23 that tries to minimize the amount of unnecessary expansion of the sequence map under the assumption that the input sequence map is *consistent*. Recall that using the two propagation operators we can obtain a consistent sequence map in $\mathcal{O}(b \cdot |\text{dom } S| \cdot i)$ time.

**Algorithm 23**: $f_V(S)$

**input** : Sequence map $S$ and variables $V = \{v_1, \ldots, v_k\} \subset \operatorname{dom} S$
**output**: $\pi_{v_1,\ldots,v_k}(R_S)$

```
1  res ← {⟨⟩};
2  AncestorVars ← CurrentVars ← V;
   // Compute all ancestors
3  while CurrentVars ≠ ∅ do
4      v ∈ CurrentVars;
5      if ∃ : v' ∈ dom S ∖ AncestorVars : (v', v) ∈ EC then
6          AncestorVars ← AncestorVars ∪ {v'};
7          CurrentVars ← CurrentVars ∪ {v'};
8      CurrentVars ← CurrentVars ∩ {v};
   // Take all root variables that are ancestors
9  RootVars ← {v ∈ dom S :∄ v' ∈ dom S : (v, v') ∈ edgeCover(S)};
10 foreach v ∈ RootVars do
11     res ← res × ProjectedRelation_V^AncestorVars(S, v, {1 → (1, |S(v)|)});
12 return res
```

Intuitively, by assuming a consistent sequence map, we can avoid even looking at bindings for variables that do not lead to (i.e., are ancestors of) variables in $V$. Furthermore, even, for variables that lead to variables in $V$, we do not really care about actual bindings, but only that it is related to bindings of a child variable that leads to a variable in $V$. If a variable has more than one child variable that is an ancestor of a variable in $V$ (we call such a variable a *branch variable* in Algorithm 23), only then, it is necessary to ensure that only combinations of bindings contributed by each of the child variables are accepted that have a common binding for the parent.

This intuition is realized in Algorithm 23 by first marking (by inclusion in AncestorVars) in lines 2–8 all ancestors of a variable in $V$.

This set of ancestors is used in lines 9–11 to compute the projected relation of any connected component rooted (w.r.t. the edge cover) at a variable that is in $V$ or ancestor of a variable in $V$.

For each such variable, we call Algorithm 24 with a sequence of intervals containing one single interval over all bindings for that variable in $S$.

Algorithm 24 computes the projection to $V$ of the induced relation of the component rooted at $v$. However, it avoids, where possible, to compute the induced relation for the variables not in $V$. For this, the central observation is that, if a variable is neither part of $V$ nor the least common ancestor of

**Algorithm 24**: ProjectedRelation$^{A}_{V}(S, v, \mathcal{I}, \text{inLCA})$

**input** : Sequence map $S$, set of variables $V \subset \text{dom } S$, variable $v \in \text{dom } S$, sequence $\mathcal{I}$ of non-overlapping sets of intervals in order of start index

**output**: Projection of the induced relation of $S$ to $V$

```
1  ChildVars ← {v′ : (v, v′) ∈ edgeCover(S)};
2  if v ∈ V then
       // Result variable: we need to return bindings for the variable in
          the result.
3      res ← ∅;
4      foreach (s, e) ∈ do
5          for i ← s to e do
6              (n, I) ← S(v)[i]; R ← {⟨v : n⟩};
7              foreach v′ ∈ π₁(I) ∩ ChildVars do
8                  R ←
                   R × ProjectedRelation_V(S, v′, π₂(σ_{1=v}(I)), true);
9              res ← res ∪ R;
10 else if |ChildVars| > 1 then
       // Branch variable: we need to ensure that bindings from two
          branches are connected to the same binding of the branch
          variable before combining
11     res ← ∅;
12     foreach (s, e) ∈ do
13         for i ← s to e do
14             (n, I) ← S(v)[i]; R ← {⟨⟩};
15             foreach v′ ∈ π₁(I) ∩ ChildVars do
16                 R ←
                   R × ProjectedRelation_V(S, v′, π₂(σ_{1=v}(I)), true);
17             res ← res ∪ R;
18 else if |ChildVars| = 1 then
       // Skip variable: we can "skip" to the next variable.
19     v′ ∈ ChildVars;
20     if inLCA = true then
           // We care that only bindings covered by any of the intervals in
              ℐ are considered
21         CurrentInts ← ∅;
22         foreach (s, e) ∈ do
23             for i ← s to e do
24                 CurrentInts ←
                   UnionInts(CurrentInts, intervals_{v′}(S(v)[i]));
25         res ← ProjectedRelation_V(S, v′, CurrentInts, inLCA);
```

two variables in $V$, it has only a single child that is ancestor of a variable in $V$ (otherwise it would be least common ancestor of all pairs of variables in $V$ where one is contained in the sub-trees rooted at one of the children and the other in the sub-tree of the other child). For computing the projection to $V$ of the induced relation, it matters in this case not which binding of the parent is connected to which binding of the child. It only matters that a bindings of the child variable is related to *any* binding of the parent (and even that only matters if we are in a sub-tree rooted at the least-common ancestor of at least one pair of variables in $V$, otherwise we just go to the next variable, line 24). This is exploited in line 23 by computing the union over the intervals of all parent bindings. Observe, that the union of non-overlapping intervals, as computed by Algorithm 25, is still a (minimal) sequence of non-overlapping intervals and can be computed efficiently (in linear time over the two sequences of intervals). We can then continue with that single sequence of intervals. In comparison, at a branch variable (with more than one child in the ancestors $A$ of a variable in $V$) we call ProjectedRelation *for each binding*. In the worst case, the intervals associated with each binding overlap entirely and ProjectedRelation is called for each binding with an interval set that covers as many bindings of the child as the single interval set computed for a skip variable (lines 18–25).

To compute that single interval we use UnionInts, the last of the operations on sets of intervals, as shown in Algorithm 25.

**Theorem 12.20.** *Algorithm 25 computes a minimal, non-overlapping sequence of intervals representing the union of two sequences of intervals $I_1$ and $I_2$ in $\mathcal{O}(|I_1| + |I_2|)$ time and constant additional space.*

*Proof.* Algorithm 25 computes minimal, non-overlapping intervals by the same observation as for Algorithm 8: an interval is added only, if it is clear that the next binding after its end index is not covered (line 18) or if the end of the sequence is reached (line 23).

The result of Algorithm 25 are the union of the intervals in the input sequences: if a binding is covered by either interval in $I_1$ or $I_2$, then either that interval overlaps with some interval in the other sequence (line 13–16) and we extend the currently active interval by all bindings covered in these overlapping intervals (lines 20–22) or start a new interval covering these bindings (line 17–19). If it does not overlap (lines 7–12), we also either extend or create a new interval, but only for the bindings covered by the interval itself.

The algorithm relies on the fact that the interval sets are ordered by increasing start index and non-overlapping. This allows us to infer that, if an interval from $I_1$ lies before the first (remaining) interval from $I_2$, then it lies before *all* intervals of $I_2$ and vice versa. Thus deciding overlapping

**Algorithm 25**: UnionInts(Intervals$_1$, Intervals$_2$)

**input** : Two sequences Intervals$_1$, Intervals$_2$ of non-overlapping intervals (*without* associated variables) in order of start index

**output**: Minimal, non-overlapping sequence of intervals covering all indices contained in intervals of either input sequence, in order of start index

```
1  NewIntervals ← ∅ ;
2  start ← ∞; end ← 0;
3  i, j, k ← 1;
4  while i ≤ |Intervals_1| or j ≤ |Intervals_2| do
5      (v, s_1, e_1) ← Intervals_1[i] or ∞ if i > |Intervals_1|;
6      (v, s_2, e_2) ← Intervals_2[i] or ∞ if j > |Intervals_2|;
7      if e_1 < s_2 then                          // (s_1, e_1) before (s_2, e_2)
8          s ← s_1; e ← e_1;
9          i++;
10     else if e_2 < s_1 then                     // (s_2, e_2) before (s_1, e_1)
11         s ← s_2; e ← e_2;
12         j++;
13     else                                       // (s_1, e_1) overlaps (s_2, e_2)
14         s ← min(s_1, s_2);
15         e ← max(e_1, e_2);
16         i++; j++;
17     if end < s then
18         if start ≠ ∞ then NewIntervals[k] ← (start, end); k++;
19         start ← s; end ← e;
20     else
21         start ← min(start, s);     // only for illustration, always start ≤ s
22         end ← max(end, e);
23 if start ≠ ∞ then NewIntervals[k] ← (start, end); k++;
24 return NewIntervals ;
```

becomes amortised constant rather than linear in the size of the intervals.

Algorithm 25 runs in $\mathcal{O}(|I_1| + |I_2|)$ as in each iteration of the loop 4–22 either $i$ or $j$ or both are incremented. Getting the next interval from $I_1$ and $I_2$ (line 5–6) is by assumption constant, as is adding an interval at the end of NexIntervals (line 18). □

From this result, we can derive the following properties of $F$ depending on the number of variables in $V$, the number of least common ancestors of (pairs of) variables in $V$, and the size of the sub-tree rooted at such least common ancestors.

**Theorem 12.21.** *Algorithm 21 computes $F_V(S)$ for a sequence map $S$ and a set of variables $V \subset \operatorname{dom} S$ in $\mathcal{O}(b_{\mathcal{V}}^{|\mathcal{V}|} \cdot b_{\mathcal{W}}^2 + b_{\mathcal{W}}^2 + |\operatorname{dom} S|) \leq \mathcal{O}(n^2 + n^{|\mathcal{V}|} + |\operatorname{dom} S|)$ where $\mathcal{V} = V \cup \{v \in \operatorname{dom} S : \exists v' \neq v'' \in V : \operatorname{lca}_{EC}(v', v'') = v\}$ is the set of all variables in $V$ and their least common ancestors in $EC = \mathit{edgeCover}(S)$, $\mathcal{W} = \{v \in \operatorname{dom} S : \exists$ path from $v' \in \mathcal{V}$ to $v\} \setminus \mathcal{V}$ is the set of all variables in a sub-tree rooted at a variable in $\mathcal{V}$ except $\mathcal{V}$, and $b_N$ is the maximum number of bindings for a variable in a set of variables $N$ or 1 if $N = \varnothing$.*

*Proof.* In Algorithm 21, computing the ancestors of any variable in $V$ is bound by $\mathcal{O}(|\operatorname{dom} S|)$ time by stopping the mark process whenever a previously marked ancestor is found (line 5). This means, at worst each variable in dom $S$ is marked once by lines 5–7.

Algorithm 24 is called for each root of a connected component containing with variables in $V$ (determined by looking at AncestorVars).

If Algorithm 24 is called with a skip node and is in inLCA mode, it first computes a single sequence of intervals from the sequences for each bindings. This is done in $\mathcal{O}(b \cdot (i + b)) = \mathcal{O}(b^2)$ as the size of a non-overlapping sequence of intervals and thus the size of CurrentInts is bound by $b$ and we compute the union of the intervals of each of the $b$ bindings with CurrentInts. Then it calls itself once with a single sequence of intervals for the child variable of $v$. If it is not in inLCA mode, we just skip to the next variable (constant time).

On a tree, forest, or CIG data, the second interval sequence of UnionInts is always a single interval. In this case, storing the current intervals in an interval tree and querying and modifying that interval tree with each of the single intervals associated with a binding can be done in $\mathcal{O}(b \cdot \log(b))$ time rather than $\mathcal{O}(b^2)$ as the unmodified Algorithm 25. However, in the general case, an interval tree based algorithm performs worse, as the interval sequence associated with each binding is already of size $i \leq b$ and thus the overall time is $\mathcal{O}(b^2 \cdot \log(b))$.

If Algorithm 24 is called with a branch or result variable, it calls itself once for each child variable and interval set associated with a binding of $v$.

Each such interval set is bound by $b$ and the number of calls are bound by $b \times b_{\text{EC}}$ where $b_{\text{EC}}$ is the maximum degree of the edge cover of $S$.

Overall, there are $|\mathcal{V}|$ branch and result nodes and thus the overall run time is bound by $\mathcal{O}(b_{\mathcal{V}}^{|\mathcal{V}|} \cdot b_{\mathcal{W}}^2 + b_{\mathcal{W}}^2)$. □

If $V = \{v\}$ for some variable $v$, then we can omit the computation of the ancestors and the call to project relation and directly return the set of all $\text{binding}(S(v)[i])$ for $i \leq |S(v)|$. This yields an algorithm linear in the number of bindings for $v$.

For $|V| > 1$, however, the algorithm needs to know the lca's of each pair of variables in $V$ and the path from these lca's to a variable in $V$. There is a wealth of well-established approaches for finding the least common ancestor [121, 8] that, at a linear pre-processing cost for the tree (here the query), allow constant look-up of the least common ancestor. However, since we also need the *path* between $\text{lca}(v, v')$ and $v$, $v'$, the gains by adopting such an approach are limited.

With $\mathcal{F}$ the set of operators on sequence maps is complete. The following sections conclude the discussion of the operators by highlighting some of their properties.

## 12.8 ALGEBRAIC EQUIVALENCES

The sequence map operations defined above as part of the CIQCAG algebra mirror, where possible, closely relational algebra expressions. This is reflected not only in the definitions throughout the previous sections which reduce most of the operations to operations on the induced relations of the involved sequence maps, but also in the algebraic properties that govern these operations. In the following, we briefly summarize and compare the most important algebraic laws that govern the the sequence map operations. Particularly, we consider the effect of consistent and inconsistent operator variants and the propagation operators.

In the following, we denote with $\text{EC}(S)$ the edge cover of a sequence map $S$, $\textbf{sm}(R)$ a sequence map with the induced relation $R$, and $S_\varnothing = \textbf{sm}(\varnothing)$ a sequence map with induced relation $R_{S_\varnothing} = \varnothing$. For brevity, when there are both consistent and inconsistent variants of an operator that exhibit the same laws, we denote that with $^{(\lightning)}$. Note, however, that either all, or none of the operators in each formula are of the consistent variant.

NEUTRAL AND ABSORBING ELEMENT LAWS. For $\ddot{\cup}$ and $\ddot{\setminus}$, any single-variable or single-edge sequence map with induced relation $\varnothing$ is a the *neutral element*. Note, that there are, as for most other relations, many

| | | | |
|---|---|---|---|
| (N1) | $S \mathbin{\ddot{\cup}} S_\varnothing \leftrightarrow S^*$ | (N2) | $S \mathbin{\ddot{\cup}} S \leftrightarrow S$ |
| (N3) | $S \mathbin{\ddot{\curlyvee}} S_\varnothing \leftrightarrow S^*$ | | |
| (N4) | $S\ \theta\ S_\varnothing \leftrightarrow S_\varnothing{}^\dagger$ | (N5) | $S\ \theta\ S \leftrightarrow S$ |

* Precondition: $\textsc{ec}(S_\varnothing) = \textsc{ec}(S)$, both single-variable or both single-edge

† Precondition: $\textsc{ec}(S_\varnothing) \cap \textsc{ec}(S) = \varnothing$

Table 61. Neutral and absorbing elements for combination operators ($\theta \in \{\ddot{\bowtie}^{(\lightning)}_{\cap}, \ddot{\bowtie}^{(\lightning)}, \ddot{\rtimes}^{(\lightning)}\}$)

sequence maps that represent $\varnothing$ as long as we allow inconsistent (where failure markers alone give $\mathsf{Nodes}(D)^{\mathrm{dom}\, S}$ variations of a consistent sequence map that represent the same induced sequence) or not interval-minimal (where non minimal interval sets give up to $2^{\mathsf{Nodes}(D)}$ variants) sequence maps. For $\ddot{\bowtie}^{\lightning}_{\cap}$, $S_\varnothing$ forms is an absorbing element. Table 61 summarizes the laws for neutral and absorbing elements. We can also observe, that $\ddot{\bowtie}^{(\lightning)}$ and $\ddot{\cup}$ return the unmodified input sequence map if it is combined with itself.

COMMUTATIVE, ASSOCIATIVE, AND DISTRIBUTIVE LAWS. Figure 62 summarizes commutative, associative, distributive, and de-Morgan laws for sequence map combination operators: $\ddot{\cup}$ and all join variants are commutative and associative as is easy to see from their definition (C1–C3, A1–A3). Moreover, a variation (DM1–DM2) of de-Morgan laws hold for $\ddot{\curlyvee}$, $\ddot{\cup}$, and $\ddot{\bowtie}^{(\lightning)}$ (which takes the place of $\cap$ in usual set-theoretic formulations of de-Morgan laws), but only if all involved sequence maps are either all single-variable or all single-edge and have the same edge cover. This is, in fact, the precondition for all laws involving $\ddot{\curlyvee}$ or $\ddot{\cup}$. Recall, that in this case $\ddot{\bowtie}^{(\lightning)}$ is equivalent to intersection as all variables and edges are shared. $\ddot{\bowtie}^{(\lightning)}_{\cap}$ can not be used, here, as it prohibits overlapping edge covers (which are required by $\ddot{\cup}$ and $\ddot{\curlyvee}$). For the same reason, only $\ddot{\bowtie}^{(\lightning)}$ and $\ddot{\cup}$ distribute over each other (D1–D2) on all single-edge or all single-variable sequence maps, but not $\ddot{\bowtie}^{(\lightning)}_{\cap}$. Finally, distribution between the two join types $\ddot{\bowtie}^{(\lightning)}_{\cap}$ and $\ddot{\bowtie}^{(\lightning)}$ is limited, $\ddot{\bowtie}^{(\lightning)}_{\cap}$ distributes over $\ddot{\bowtie}^{(\lightning)}$ (D3), but, in general, not the other way around.

*Table 62.* The commutative and associative laws are easy to verify in the respective definition. For the de-Morgan laws, consider that a tuple is in the induced relation of $S_1 \mathbin{\ddot{\curlyvee}} (S_2 \mathbin{\ddot{\cup}} S_3)$ if it is in $R_{S_1}$ but neither in $R_{S_2}$ nor in $R_{S_3}$. It is in $(S_1 \mathbin{\ddot{\curlyvee}} S_2) \mathbin{\ddot{\bowtie}^{(\lightning)}} (S_1 \mathbin{\ddot{\curlyvee}} S_3)$ if it is both in $R_{S_1}$ but not in $R_{S_2}$ and

| | |
|---|---|
| (C1) | $S_1 \mathbin{\ddot{\cup}} S_2 \leftrightarrow S_2 \mathbin{\ddot{\cup}} S_1$ * |
| (C2) | $S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_2 \leftrightarrow S_2 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_1$ † |
| (C3) | $S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_2 \leftrightarrow S_2 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_1$ |
| (A1) | $(S_1 \mathbin{\ddot{\cup}} S_2) \mathbin{\ddot{\cup}} S_3 \leftrightarrow S_1 \mathbin{\ddot{\cup}} (S_2 \mathbin{\ddot{\cup}} S_3)$ * |
| (A2) | $(S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_2) \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_3 \leftrightarrow S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} (S_2 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_3)$ † |
| (A3) | $(S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_2) \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3 \leftrightarrow S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} (S_2 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3)$ |
| (DM1) | $S_1 \mathbin{\ddot{\setminus}} (S_2 \mathbin{\ddot{\cup}} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\setminus}} S_2) \mathbin{\ddot{\bowtie}}^{(\lightning)} (S_1 \mathbin{\ddot{\setminus}} S_3)$ * |
| (DM2) | $S_1 \mathbin{\ddot{\setminus}} (S_2 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\setminus}} S_2) \mathbin{\ddot{\cup}} (S_1 \mathbin{\ddot{\setminus}} S_3)$ * |
| (D1) | $S_1 \mathbin{\ddot{\cup}} (S_2 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\cup}} S_2) \mathbin{\ddot{\bowtie}}^{(\lightning)} (S_1 \mathbin{\ddot{\cup}} S_3)$ * |
| (D2) | $S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} (S_2 \mathbin{\ddot{\cup}} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_2) \mathbin{\ddot{\cup}} (S_1 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3)$ * |
| (D3) | $S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} (S_2 \mathbin{\ddot{\bowtie}}^{(\lightning)} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_2) \mathbin{\ddot{\bowtie}}^{(\lightning)} (S_1 \mathbin{\ddot{\bowtie}}_{\cap}^{(\lightning)} S_3)$ ‡ |
| (D4) | $S_1 \mathbin{\ddot{\ltimes}}^{(\lightning)} (S_2 \mathbin{\theta} S_3) \leftrightarrow (S_1 \mathbin{\ddot{\ltimes}}^{(\lightning)} S_2) \mathbin{\theta} (S_1 \mathbin{\ddot{\ltimes}}^{(\lightning)} S_3)$ ‡ |

* Precondition: $\text{EC}(S_1) = \text{EC}(S_2) = \text{EC}(S_3)$, $S_1, S_2, S_3$ all single-variable or all single-edge

† Precondition: $\text{EC}(S_1) \cap \text{EC}(S_2) = \varnothing$, $\text{EC}(S_1) \cap \text{EC}(S_3) = \varnothing$, $\text{EC}(S_2) \cap \text{EC}(S_3) = \varnothing$

‡ Precondition: $\text{EC}(S_1) \cap \text{EC}(S_2) = \varnothing$, $\text{EC}(S_1) \cap \text{EC}(S_3) = \varnothing$

Table 62. Commutative, associative, distributive, de Morgan laws ($\theta \in \{\ddot{\bowtie}_{\cap}^{(\lightning)}, \ddot{\bowtie}^{(\lightning)}\}$)

| | | | |
|---|---|---|---|
| (S1) | $\ddot{\sigma}_c^{(\not)}(\ddot{\sigma}_c^{(\not)}(S)) \leftrightarrow \ddot{\sigma}_c^{(\not)}(S)$ | | |
| (S2) | $\ddot{\sigma}_c^{(\not)}(S_1\ \theta\ S_2) \leftrightarrow \ddot{\sigma}_c^{(\not)}(S_1)\ \theta\ \ddot{\sigma}_c^{(\not)}(S_2)^{*}$ | (S2a) | $\ddot{\sigma}_c^{(\not)}(S_1\ \phi\ S_2) \leftrightarrow \ddot{\sigma}_c(S_1)\ \phi\ \ddot{\sigma}_c(S_2)^{*}$ |
| (S3) | $\ddot{\sigma}_c^{(\not)}(S_1\ \theta\ S_2) \leftrightarrow \ddot{\sigma}_c^{(\not)}(S_1)\ \theta\ S_2{}^{\dagger}$ | (S3a) | $\ddot{\sigma}_c^{(\not)}(S_1\ \phi\ S_2) \leftrightarrow \ddot{\sigma}_c(S_1)\ \phi\ S_2{}^{\dagger}$ |
| (S4) | $\ddot{\sigma}_c^{(\not)}(S_1\ \theta\ S_2) \leftrightarrow S_1\ \theta\ \ddot{\sigma}_c^{(\not)}(S_2)^{\ddagger}$ | (S4a) | $\ddot{\sigma}_c^{(\not)}(S_1\ \phi\ S_2) \leftrightarrow S_1\ \phi\ \ddot{\sigma}_c(S_2)^{\ddagger}$ |

* Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_1 \cap \operatorname{dom} S_2$

† Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_1 \setminus \operatorname{dom} S_2$

‡ Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_2 \setminus \operatorname{dom} S_1$

Table 63. Selection laws ($\theta \in \{\ddot{\cup}, \ddot{\bowtie}_{\cap}^{\not}, \ddot{\bowtie}^{\not}, \ddot{\ltimes}^{(\not)}, \ddot{\setminus}\}$, $\phi \in \{\ddot{\bowtie}_{\cap}, \ddot{\bowtie}\}$)

in $R_{S_1}$ but not in $R_{S_3}$. This is the case due to the precondition that the edge covers of all three sequence maps are the same. Thus, $R_{S_1} \bowtie R_{S_2} = R_{S_1} \cap R_{S_2}$. Analog for DM2.

For the distributive laws D1 and D2, the core observation is again that all involved sequence maps have the same edge cover. Thus the induced relation of $\ddot{\bowtie}^{(\not)}$ is the intersection of the induced relations of its input sequence maps.

For D3, consider that $S_1\ \ddot{\bowtie}_{\cap}^{(\not)}\ (S_2\ \ddot{\bowtie}^{(\not)}\ S_3)$ is a valid expression only if $\textsc{ec}(S_2\ \ddot{\bowtie}^{(\not)}\ S_3) \cap \textsc{ec}(S_1) = \textsc{ec}(S_2) \cup \textsc{ec}(S_3) \cap \textsc{ec}(S_1) = \varnothing$. Thus it is valid if $\textsc{ec}(S_1) \cap \textsc{ec}(S_3) = \varnothing$ and $\textsc{ec}(S_2) \cap \textsc{ec}(S_3) = \varnothing$, in which case also $(S_1\ \ddot{\bowtie}_{\cap}^{(\not)}\ S_2)\ \ddot{\bowtie}^{(\not)}\ (S_1\ \ddot{\bowtie}_{\cap}^{(\not)}\ S_3)$ is valid. □

SELECTION. Selection is generally a good candidate for optimization, pushing selections (a fast but possibly fairly selective operation) inside of an expression thus limiting the size of intermediary results. Also selection can generally be propagated "down" into an expression: Selection distributes over $\ddot{\cup}$, $\ddot{\bowtie}_{\cap}^{\not}$, $\ddot{\bowtie}^{\not}$ and $\ddot{\setminus}$, see Table 63. For $\ddot{\bowtie}_{\cap}$ and $\ddot{\bowtie}$, we need to ensure that the consistency of the input sequence maps is retained and thus can only push $\ddot{\sigma}$ inside (regardless of the outer selection variant), cf. (S2a, S3a, S4a). For $\ddot{\setminus}$ and $\ddot{\ltimes}^{(\not)}$, we can actually drop the selection around $S_2$ in (S2), since we only retain tuples from the first input sequence map.

PROJECTION. In contrast to selection, we can only propagate selection "down" in an expression to the point where the projection condition that $V \subset \operatorname{dom} S$ and, for all pairs $v, v' \in V$ all variables on the path from $\operatorname{lca}(v, v')$ to each variable are in $V$ still hold in the sub-expressions. Conversely, an expression might benefit from introducing additional projec-

$$(\text{P1}) \quad \ddot{\pi}_V(\ddot{\pi}_{V'}(S)) \leftrightarrow \ddot{\pi}_{V \cup V'}(S)^{*} \qquad (\text{P2}) \quad \ddot{\pi}_V(\ddot{\sigma}_c^{(\prime)}(S)) \leftrightarrow \ddot{\sigma}_c^{(\prime)}(\ddot{\pi}_V(S))^{\dagger}$$

$$(\text{P3}) \quad \ddot{\pi}_V(S_1 \;\theta\; S_2) \leftrightarrow \ddot{\pi}_{V_1}(S_1) \;\theta\; \ddot{\pi}_{V_2}(S_2)^{\ddagger} \qquad (\text{P4}) \quad \ddot{\pi}_V(S_1 \;\ddot{\ltimes}^{(\prime)}\; S_2) \leftrightarrow \ddot{\pi}_{V_1}(S_1) \;\ddot{\ltimes}^{(\prime)}\; S_2{}^{\ddagger}$$

* Precondition: the projection condition holds for $V, V'$ and no pair of variables from $v \in V$ and $v' \in V'$ has a common ancestor in the edge cover of $S$

† Precondition: $\mathsf{Vars}(c) \notin V$

‡ Precondition: $V_1 \cup V_2 = V$, $V_1$ ($V_2$) fulfills the projection condition for $S_1$ ($S_2$) with $\mathrm{dom}\, S_1 \cap V \setminus V_1 = \varnothing$ ($\mathrm{dom}\, S_2 \cap V \setminus V_2 = \varnothing$)

Table 64. Projection laws ($\theta \in \{\ddot{\bowtie}_{\cap}^{(\prime)}, \ddot{\bowtie}^{(\prime)}\}$)

$$(\varpi 1) \quad \ddot{\omega}_v^{\blacktriangle}(S) \leftrightarrow S^{*} \qquad (\varpi 2) \quad \ddot{\omega}_v^{v}(S) \leftrightarrow S^{\dagger}$$

$$(\varpi 3) \quad \ddot{\omega}(S_1 \;\theta\; S_2) \leftrightarrow S_1 \;\theta\; S_2{}^{\ddagger} \qquad (\varpi 4) \quad \ddot{\omega}(\ddot{\sigma}_c(S)) \leftrightarrow \ddot{\sigma}_c(S)^{\S}$$

$$(\varpi 5) \quad \ddot{\omega}(\ddot{\mu}_v(D, Q, R)) \leftrightarrow \ddot{\mu}_R(D, Q)v \qquad (\varpi 6) \quad \ddot{\omega}(\ddot{\mu}_{v_1, v_2}(D, Q)) \leftrightarrow \ddot{\mu}_{v_1, v_2}(D, Q)$$

* Precondition: $v \in \mathrm{dom}\, S$ equivalence does not preserve consistency

† Precondition: $v \in \mathrm{dom}\, S$ equivalence does not preserve consistency

‡ Precondition: $S_1, S_2$ consistent

§ Precondition: $S$ consistent

Table 65. Propagation laws ($\ddot{\omega} \in \{\ddot{\omega}_v^{\blacktriangle}, \ddot{\omega}_v^{v}\}$, $\theta \in \{\ddot{\setminus}, \ddot{\bowtie}_{\cap}, \ddot{\bowtie}, \ddot{\ltimes}\}$)

tions to get rid of attributes not used in the remainder of an expression as early as possible, viz. immediately after the innermost expression referencing them.

Given two sets of variables $V$ and $V'$ such that the projection condition on $S$ holds for both sets and no pair $v \in V, v' \in V'$ has a common ancestor in the edge cover of $S$, we can combine a sequence of projections for these two sets into a single projection for $V \cup V'$ (P1). Projection and selection can be arbitrarily ordered as long as $\mathsf{Vars}(c) \notin V$ (P2). Finally, $\ddot{\pi}$ distributes over $\ddot{\bowtie}_{\cap}^{(\prime)}$, $\ddot{\bowtie}^{(\prime)}$, $\ddot{\ltimes}^{(\prime)}$ if there are subsets $V_1$, $V_2$ of $V$ such that each subset affects one of the input sequence maps but not the other. For $\ddot{\ltimes}^{(\prime)}$, we can omit the projection on $S_2$ since only bindings from $S_1$ are retained.

PROPAGATION. Both propagation operators do not affect the induced sequence only the consistency state. Thus, as long as we are only interested in the induced sequence, we can add or remove propagation operators arbitrarily ($\varpi 1$–$\varpi 2$). Finally, for each of the consistent sequence map operators

(such as $\ddot{\bowtie}_{\cap}$ or $\ddot{\searrow}$) we can drop any surrounding propagation operators if the input sequence maps are consistent itself ($\bar{\omega}3$–$\bar{\omega}6$).

EXTRACT. Extract returns a relation rather than a sequence map. Thus it can not be distributed over any of the sequence map operations. However, it may interact with relational expressions as discussed in Chapter 13.

This concludes the brief overview of the most important algebraic laws for sequence map operators in CIQCAG. Before we extend the sequence map operators that are limited to the evaluation of tree queries, to the full CIQCAG algebra with (standard relational) operators for the evaluation of non-tree parts of an arbitrary query and operators for construction and iteration, we briefly outline in the following section an iterator model for the physical realisation of the sequence map operators.

## 12.9 ITERATOR IMPLEMENTATION

Iterator or stream model has proved essential for the scalable evaluation of relational queries, see, e.g., [114]. Here, we briefly outline how CIQCAG's sequence map operators can be implemented in an iterator model that reduces the space complexity of the evaluation compared to the sequence-at-a-time model discussed above. This holds in particular on tree or forest data and if the number of variables that are either part of the answer or used in the non-tree part of the query (i.e., the number of variables extracted by $_F$) is small compared to the full number of variables in the query.

Let $_{F_V}(\mathfrak{E})$ be a CIQCAG expression such that $\mathfrak{E}$ contains only join, semi-join, selection and initialization operators. We limit the operators to keep the discussion brief, but we can extend the approach to allow also projection, union, and difference. Note, that we can order $\mathfrak{E}$ in such a way that all unary conditions (i.e., the unary initialization operators and joins to connect them) for each query variable $v$ are clustered and the result of all these unary conditions is connected by a single join to the expressions representing the conditions on each child variable of $v$. We denote with $\mathcal{L}(v)$ the unary conditions relating to $v$ and with $\mathcal{E}(v)$ the full expression for $v$ including the expressions for its children.

Consider the query in Figure 64. It can be realized by the CIQCAG expres-

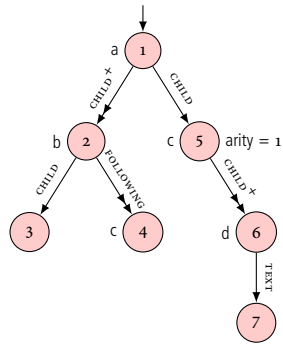



Figure 64. Example query for iterator approach

sion $F_{v_7}(\mathfrak{E})$ where

$$\mathfrak{E} = \overbrace{\ddot{\mu}_{v_1}(D, Q, \text{Label}_{\text{'a'}})}^{\mathcal{L}(v_1)} \ddot{\bowtie} \ddot{\mu}_{v_1,v_2}(D, Q) \ddot{\bowtie} \ddot{\mu}_{v_1,v_5}(D, Q)$$
$$\ddot{\bowtie} \ddot{\mu}_{v_2}(D, Q, \text{Label}_{\text{'a'}}) \quad \Big\} \mathcal{E}(v_2)$$
$$\ddot{\bowtie} \ddot{\mu}_{v_2,v_3}(D, Q)$$
$$\ddot{\bowtie} \ddot{\mu}_{v_2,v_4}(D, Q) \ddot{\mu}_{v_4}(D, Q, \text{Label}_{\text{'c'}}) \quad \Big\}$$
$$\ddot{\bowtie} \ddot{\mu}_{v_5}(D, Q, \text{Label}_{\text{'c'}}) \ddot{\bowtie} \ddot{\mu}_{v_5}(D, Q, \text{Arity}_{=1}) \quad \Big\} \mathcal{E}(v_5)$$
$$\ddot{\bowtie} \ddot{\mu}_{v_5,v_6}(D, Q) \ddot{\bowtie} \ddot{\mu}_{v_6}(D, Q, \text{Label}_{\text{'d'}})$$
$$\ddot{\bowtie} \ddot{\mu}_{v_6,v_7}(D, Q) \quad \Big\} \mathcal{E}(v_1)$$

This is an example of the general shape of an expression such that it is amenable to the iterator approach discussed in this section. Figure 65 illustrates this shape in detail: blue colored parts of the query together with their root represent the part of the query rooted at the corresponding variable $v$, i.e., $\mathcal{E}(v)$. The yellow colored parts represent the unary conditions for each variable, i.e., $\mathcal{L}(v)$. For each query variable $v$, there is a *representing node* (indicated by ● and labeled with the variable in Figure 64). Under this node we find, in general, two children: one for $\mathcal{L}(v)$ and one join node grouping all of $v$'s children (we use here a multi-way join instead of a sequence of binary joins to simplify the presentation). For each child variable $v'$ of $v$, this node has a *connection node* $\ddot{\mu}_{v,v'}(D, Q)$ and the representative join node for $v'$. Some variables have no unary restrictions, some no children in which case the respective parts are omitted (see $v_4$). If a variable has neither, we use the connection node as representative node for that variable (see $v_3, v_7$).

Based on this ordering of the `ClQcAG` expression, we can now define a processing scheme that computes the resulting sequence map for the

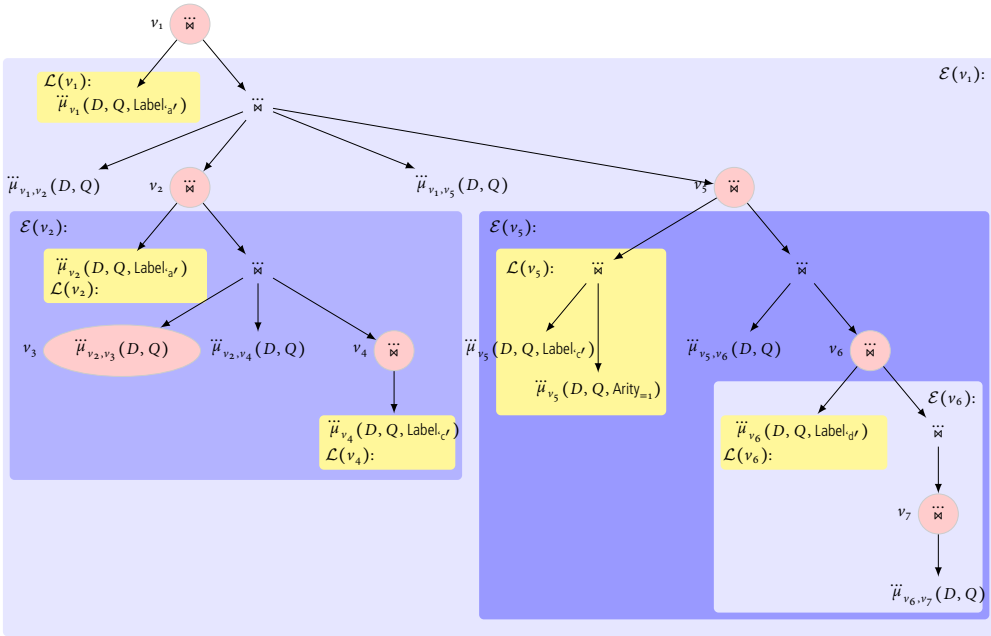



Figure 65. Operator tree for query from Figure 64

entire expression incrementally, outputs potential answers as soon as there is an extension to a full answer, and discards potential answers as soon as there can be no more extension to a full answer.

*Outline of the iterator processing scheme*

The fundamental observation for this scheme is that all unary operators are already implemented in an iterator fashion, see Section 12, i.e., as a single pass over the input sequence using, for each binding in the sequence, only conditions on that binding itself (and not on any other binding) to determine its inclusion or exclusion from the result.

For $\ddot{\bowtie}$ and $\ddot{\mu}_{v,v'}(D, Q)$, this is not so easy to see from the Algorithms in Section 12. In the following, we focus on these two operators to detail the incremental processing scheme. First, we annotate each join with the intersection of the variables of the sequence maps computed by its children. For all variable nodes in Figure 65, this yields a single variable, viz. the one shown as label in Figure 65. For the semi-joins (which only connect unary operators), this yields also a single variable (viz. the variable their parent is annotated with). For child joins, i.e., joins with one $\ddot{\mu}_{v,v'}(D, Q)$ and, possibly, one variable node, this yields the two variables $v, v'$ (note that the multi-way joins in Figure 65 are realized as a sequence of binary joins) such that $v = \mathsf{parent}(v')$.

The first change is that an operator no longer computes the entire result sequence map at once, but that each operator provides an interface with three functions next, out, and close. This extends the iterator interface used, e.g., in [114] for relational operators with out and modifies the semantics for next: We can call next on each operator with or without a pair (variable, binding).

*Unary initialization*, if called without such a pair, compute the next binding in their result sequence and return that binding (or **false** if there are no more bindings), remembering the point in the result sequence reached by the computation as well as the computed partial sequence of bindings; if called with such a binding pair they return the binding, if it is included in the result sequence and the variable is consistent. They also advance the computation of that result sequence up to the point where the binding is computed, if necessary (i.e., if the binding has not already been computed). Otherwise they return **false**.

A join remembers the *all* bindings returned by its first operand as well as the last pair next was called with. If next is called on a *join* and the passed pair is the same as in the last call, it calls the right operand's next with the last remembered pair returned by the first operand. If that call returns **false** or the passed pair is different from the last call, it calls the left operand's next with the passed pair, if there is any, and returns **false** if that call returns **false**. Otherwise, it remembers the new pair returned and calls next on the right operand with that pair. If that returns **false**, we loop and call next on

the left operand again etc. As soon as the right operand's next returns a pair, that pair is returned.

For *binary initialization* operators for $v, v'$, we return for a call with next **(1)** if a pair $(v'', n)$ is passed with $v'' = v$, and $n$ is in the partial sequence of bindings for $v$ already computed, we return the next binding for $v'$ related to $n$ (i.e., covered by an interval pointer associated with $n$) together with $v'$, if there is none return **false**. Otherwise, we expand the partial sequence of bindings for $v$ until we find $n$ or the first binding $n'$ with $n > n'$ wrt. the order associated to the relation between $v$ and $v'$. If we find $n$, return the first binding for $v'$ related to $n$, if there is any, together with $v'$. Otherwise return **false**. **(2)** if no pair is passed, we take the first binding $n$ for $v$ where not all related bindings for $v'$ have been generated (and return **false** if there is no such binding). For that binding, we compute the next binding for $v'$ and return that binding. Note, that the binary initialization stores the partial binding sequences for $v$ and $v'$ as well as all interval pointers from bindings in the partial binding sequence for $v$ to bindings of $v'$ for which not all covered bindings have been returned by next. For each such binding $n$ in the partial binding sequence of $v$, we also store a single pointer into the sequence of bindings for $v'$ indicating the last returned binding for $v'$ that is related to $n$.

The second function close removes from all operators the current state, i.e., resets the iterations (as for the iterator approach on relational operators discussed in [114]).

The final function of the iterator interface, out, each operand calls out on its child operators. In addition, if an operator is a variable node, it outputs the current binding for the variable it is labeled with together with any intervals associated with it.

The query is embedded in an outer loop, that calls next on the first operand of the query (for queries with size > 1 always a join). If the call returns **true**, it calls **out** on that join and continues the loop. Otherwise it terminates.

*Processing Figure 65*

Consider again the operator tree from Figure 65: First next is called on the top-level join. The join calls its first operand $(\ddot{\mu}_{v_1}(D, Q, \text{Label}_{\text{'a'}})$ without parameter. $\ddot{\mu}_{v_1}(D, Q, \text{Label}_{\text{'a'}})$ returns the first node in the document with label 'a'. With that binding for $v_1$ the top-level join calls the "child join". That join calls first $\ddot{\mu}_{v_1,v_2}(D, Q)$ with the binding pair and then with the result the variable join node for $v_2, \ldots$ Two observations are crucial here: Except for the leftmost, all initialization operators are always called with a binding pair. Partial sequence maps are stored in initialization operators (joins only retain the last successful binding from the second operand). E.g., $\ddot{\mu}_{v_2,v_3}(D, Q)$ builds a sequence map consisting of bindings for $v_2$ in the associated order of the relation between $v_1$ and $v_2$, and bindings for $v_3$

in the associated order of the relation between $v_2$ and $v_3$. Only bindings for $v_2$ are "restricted" by binding pairs passed in next, not bindings for $v_3$. This avoids dropping intermittent bindings for $v_3$ and thus allows to keep the interval pointers as given by the interval representation discussed in Section 12.2. Each of these sequence maps is bound by $\mathcal{O}(n \times i)$ where $n$ is the number of nodes in $D$ and $i$ the maximum size of an interval per binding. For join operators that are also variable nodes, we also store a sequence map over the bindings of the variable the node is labeled with. These sequence map are bound by $\mathcal{O}(n)$ since we do not retain intervals and can, at no additional cost, be ordered in the associated order of the incoming edge of the variable the join is labeled with.

### 12.9.1 OPTIMAL SPACE BOUNDS FOR TREE DATA

*Tighter space limits for tree data*

In contrast to the operators discussed in Section 12, here we compute sequence maps incrementally. For tree data, we can profit from another regularity to further decrease the space complexity: Recall, that tree data corresponds to relations with image disjointness property and, if we also allow closure axis, with image containment property. For a relation with image disjointness property, the number of parents of a node is bound by 1. In other words, as soon as we have found a single parent for a node, there can be no further nodes related to it among the bindings for the parent variable. For a relation $R$ with image containment, the number of parents is limited by the depth of the forest represented by the relation whose closure is $R$ (by Theorem 11.4 there is such a forest-shaped base relation for each $R$). Consider, e.g., XPath closure relations descendant, following, or following-sibling. For descendant the base relation is child and as such the number of parents under descendant is limited by the depth of the queried XML tree. For following, the base relation is the relation associating with each element the next element in document order. For that relation, the number of ancestors is limited only by the size of the XML tree. For following-sibling it is the relation that associates with each node the next element in document order that has the same parent. Thus, the number of ancestors is limited by the degree of the XML tree.

How can we exploit this observation in the iterator algorithm sketched above? The aim is to reduce the size of the partial sequence maps and the stores in join operators by deleting nodes as soon as they can no longer contribute to any further match under the assumption that all relations carry image containment or image disjointness property. We know, that as soon as a binding is related to 1, resp. $d$ (depth of base relation), different bindings of its parent, it can not contribute to any further and, thus no

longer needs to be stored. However, for image containment the related bindings may scattered over the entire binding sequence for the parent variable and thus a binding added at the beginning of the processing (i.e., related to the first binding of the parent variable) may be amenable to be removed only at the very end (if the last binding of the parent variable is also related to it).

*From individual relations to consistent orders*

Thus, we impose a further property to hold for the relations used in a query, that guarantees us that all parent bindings related to a given child binding are "clustered" together. It also ensures, that there are no parent bindings $n$ that are related to child bindings prior to child bindings related to parent bindings prior to $n$:

**Definition 12.14** (Order-compatible query)**.** Let $D$ be a relational structure and $Q$ a query on $D$. Then $Q$ is called *order-compatible* if, for each pair $(v, v')$ of parent-child variables in $Q$ with $\mathsf{rel}(v) = R$ and $\mathsf{rel}(v') = R'$, it holds that $n <_R n'$ implies that, for all nodes $c \in R'(n), c' \in R'(n')$, $c, c' \in R'(n) \cap R'(n')$ or $c \leq_{R'} c'$.

Intuitively, the two orders associated with the parent and the child variable are compatible in the sense that if a given child binding is either related to a binding of the parent or neither it or any child bindings after the given one are related to a binding of the parent.

The class of order-compatible queries is interesting because of the following result:

**Theorem 12.22.** *For a tree query $Q$ containing only a single, forest-shaped base relation $R$ and its closure relation $C$ there are orders for $R$ and $C$ such that $Q$ is* order-compatible *and the images of all nodes under $R$ and $C$ form a single continuous interval under the respective orders.*

*Proof.* In the following, we consider only tree based relations for simplicity. However, for forest we simply add some fix order on the connected components of the forest to the definitions.

Order the children of each node in $R$ in some order and call the ordered tree induced by $R$ and this order $T$. Choose as order for $R$ the breadth-first left-to-right preorder traversal $<_b$ of $T$. Choose as order for $C$ the depth-first left-to-right preorder traversal $<_d$ of $T$.

Then, if $n <_b n'$, either $n$ ancestor of $n'$ ($<_b$ is *preorder*) and $R'(n') \subset R'(n)$ or there is an ancestor $a$ of $n$ and an ancestor $a'$ of $n'$ such that $a$ is a preceding sibling of $a'$ (and both are children of $\mathrm{lca}(a, a')$). Then, also $a <_c a'$ (since the depth-first traversal is *left-to-right*) and all descendants of $a$ come before all descendants of $a'$ in $<_c$. In particular any $c \in C(n)$ is a descendant of $a$, any $c' \in C(n')$ a descendant of $a'$ and thus $c <_c c'$. □

This yields, e.g., that tree queries containing only child and descendant.

**Corollary 12.3.** *Tree queries containing only* child *and* descendant *are* order-compatible *(for some orders on* child *and* descendant*, resp.).*

*Pruning for the iterator scheme*

Under these assumptions (tree or forest data, order-compatible query wrt. the associated orders of the involved relations), we can now adapt the algorithm: **(1)** As soon as we find no more child bindings for a parent, that parent is dropped from the sequence map of a binary initialization operator, since the data is tree shaped and thus each parent has a single interval pointer. **(2)** Each child binding is dropped as soon as the first parent binding is encountered, that does not relate to it, since no further parent binding can related to that child binding. **(3)** When next is called for a binary initialization operator, we alternate between the two binding sequences: We find the first binding for the parent together with its first related binding for the child. The next call to next returns not the next binding for the child variable related to current parent binding (as in the original algorithm), but the next parent binding related to the current child binding. If there is no such binding left, we delete the child binding and continue with the first binding of the parent variable that has yet more related bindings for the child variable (and delete all parents before that parent binding). We let the binary initialization operator report the deleted bindings (as pairs of variable and the largest deleted bindings) as a further result of next to its parent (join) operator. **(4)** For unary initialization operators, we only store the last binding. **(5)** In a join operator, we delete all binding up to and including the delete bindings returned by a next and propagate them upwards, unless they are for the variable the join is labeled with. If the left operand of the join operator is no binary initialization operator, that join operator only stores the last binding.

**Theorem 12.23.** *For order-compatible queries on tree or forest data, the above algorithm runs in $\mathcal{O}(q \cdot n + o)$ time and $\mathcal{O}(q \cdot d + o)$ space where $o$ is the size of the output, $q$ is the size of the query, $n$ the size of the data, and $d$ the depth of the tree or forest.*

*Proof.* The modifications of the algorithm do not affect correctness. For the binary initialization operators this follows from the forest shape of the data and the order-compatibility of the query. For the join operators without left binary initialization operator, there is either always an ancestor join operator that memoizes the bindings (e.g., for the "child join" operators in Figure 65) or it is the top-level join. For unary initialization operators, we can observe that same fact. The reason this holds is that the query is connected and tree-shaped. Thus for each variable except for the root variable there is a binary initialization operator (and a corresponding

join) that "generates" bindings for the variable from bindings of the parent variable.

The time complexity is the same as for the sequence-at-a-time operators from Section 12. Note that we operate on tree or forest data and thus the number of intervals per binding is at most 1.

It retains the complexity of the set-at-a-time algorithm by memoizing already computed bindings in variable join nodes and initialization operators, as described above. The deletion of bindings does not affect the complexity (it adds an additive factor of $n$ to the complexity).

However, by alternating between parent and child nodes we can ensure that all the partial binding sequences in the sequence maps of the binary initialization operators contain at most $d$ bindings for the parent variable and at most 1 binding for the child variable.

The memoization structures in the join operators remove bindings at the same time as their sub-ordinate binary initialization operators. Thus, they are also bound by $d$. □

**Corollary 12.4.** *Queries containing only XPath's child and descendant relation are evaluated by the above algorithm in $\mathcal{O}(q{\cdot}n{+}o)$ time and $\mathcal{O}(q{\cdot}d{+}o)$ space where $o$ is the size of the output, $q$ is the size of the query, $n$ the size of the data, and $d$ the depth of the tree or forest.*

The last results are optimal wrt. data complexity as they coincide with the $\Omega(d + o)$ lower bound for the data complexity of such queries shown in [190].

However, it is an open question, whether order-compatible queries are the largest class of queries that can be evaluated with this complexity.

# 13

# CIQCAG: GRAPH QUERIES WITH COMPLEX HEADS



In the previous chapters, we have highlighted how CIQCAG handles tree queries or tree cores of arbitrary queries: It employs a novel data structure, the sequence map, and its operations to exploit interval properties and representations of the underlying data and thus provide highly scalable (linear time and space) evaluation of tree queries for tree and many graph data sets.

Moving beyond tree queries, we turn in this chapter to arbitrary (graph-shaped) queries as defined in Chapter 6. This is complemented by a translation from CIQLog as introduced there to the CIQCAG algebra. The journey is split in two steps: first we illustrate how non-recursive, single-rule CIQLog expressions are realized in CIQCAG: a tree core is expressed using the sequence map operators discussed in the previous chapter and remaining CIQLog features (non-tree query parts and construction) are plugged on-top of that using standard relational algebra. Second, glance at full CIQLog by discussing options for adding recursion to the CIQCAG algebra. The latter part is merely an outlook, with the current prototype choosing, merely for ease of implementation, a naïve forward chaining operator with inflationary semantics.

## 13.1 GRAPH QUERIES AND MAP EXPANSION

Using the sequence map operators discussed in the previous chapters, we can express arbitrary tree queries. With the extract operator, $\mathcal{F}$, the sequence map algebra already provides a bridge from results of the tree query evaluation (represented succinctly in a sequence map) to standard

relations. Thus, we can employ standard relational operators to express any part of a query that is not part of a given tree core of the query. We employ in the following three operators in addition to those of the textbook relational algebra of, e.g., [2]. All three are common extensions provided in many practical SQL databases.

**(1)** We add a *assignment operator* that allows for sharing of query parts (in particular of the underlying tree query) on the level of the algebra. This is a common extension (cf. [2], often in conjunction) with iteration and sequencing/composition, see Section 13.4. It does not affect the expressiveness (or complexity) of the algebra. We use syntax and semantics as in [2].

**(2)** We add a *value invention operator* $\iota_{new:\langle v_1,\ldots,v_k\rangle}(R)$ that extends the relation $R$ with a new attribute *new* which contains a new value for each unique group over the grouping attributes $v_1,\ldots,v_k$. It is a slight variation of the *new* operator in [2] or languages with value invention such as ILog. In practical SQL databases it can be realized, e.g., using the numbering operator described next.

**(3)** We add a *numbering operator* $\nu_{order:\langle order',o,v_1,\ldots,v_k\rangle}(R)$ that extends the relation $R$ with a new attribute *order* such that given a sort order on integers and values for $v_1,\ldots,v_k$ the values of *order* consecutively number the groups in $R$ over the grouping attributes $v_1,\ldots,v_k$. The additional values *order*' and offset *o* serve to implement the nested order terms as described in Chapter 6. In practical databases, the numbering operator is provided in form of `DENSE_RANK` in SQL:1999 [159] or ordered relations as used in Monet [39], see also [117].

Numeric aggregation and aggregation terms are treated as usual, cf. [102], and omitted for space reasons in the following.

Before we turn to the detailed translation function from CIQLog to CIQCAG that demonstrates the use of these operators for evaluating non-tree parts of a query, let us reconsider some of the CIQLog queries from Part III and their CIQCAG equivalents.

## 13.2 TRANSLATION BY EXAMPLE

We start with a very simple example from Section 8.2, where we consider the translation of XPath. The CIQLog rule shown below expresses a (tree-shaped) query and the selection of a single variable (here $v_3$) into the answer relation:

```
1 ans(v3) ⟵ root(v1) ∧ CHILD+(v1,v2) ∧ 𝔏(v2,paper) ∧
    CHILD(v2,v3) ∧ 𝔏(v3,author)
```

In CIQCAG, the entire body can be realized using sequence map operators from Chapter 12 as it is tree shaped. Here and in the following we use sequence map operators that may introduce inconsistencies. Therefore, we have to add propagation operators before extracting the results (via $F$) from the sequence map. We can, as well, employ only consistent operators in which case we can omit the propagation.

$$
\begin{aligned}
\text{ANS} &:= \pi_{v_3}(\text{BINDINGS}) \\
\text{BINDINGS} &:= F_{v_3}(\ddot{\bar{\omega}}^{v_2}_{\downarrow}(\ddot{\bar{\omega}}^{\uparrow}_{v_3}((\ddot{\mu}_{v_1}(D, Q, \text{root}) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_1,v_2}(D, Q) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2}(D, Q, \text{'paper'})) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2,v_3}(D, Q) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_3}(D, Q, \text{'author'}))))
\end{aligned}
$$

As in the following example, we separate the evaluation of the body in a relation called BINDINGS. This allows us to reference the result multiple times.

Turning to a more complex example from Section 7.4 (Xcerpt translation), we consider a fairly complex head in the following CIQLOG rule:

```
root(id1) ∧ 𝔏(id1, authors) ∧ CHILD(id1, id2(v3)) ∧ deep-copy(v3, id2(v3))
⟵ root(v1) ∧ 𝔏(v1, conference) ∧ CHILD+(v1, v2) ∧ 𝔏(v2, paper) ∧ CHILD
    (v2, v3) ∧ 𝔏(v3, author).
```

Again the body is essentially tree shaped and can thus be realized using only sequence map operators:

$$
\begin{aligned}
\text{BINDINGS} &:= F_{v_3}(\ddot{\bar{\omega}}^{v_2}_{\downarrow}(\ddot{\bar{\omega}}^{\uparrow}_{v_3}((\ddot{\mu}_{v_1}(D, Q, \text{root}) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_1}(D, Q, \text{'conference'}) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_1,v_2}(D, Q) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2}(D, Q, \text{'paper'})) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2,v_3}(D, Q) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_3}(D, Q, \text{'author'}))))
\end{aligned}
$$

Suppose we add another relation CHILD(v1, v3) to the query. Either that new relation or the existing connection between $v_1$ and $v_3$ can not be part of the tree core of the query and thus must be realized using standard relational expressions. The main change is that we need to extract also $v_1$ from the sequence map using $F$ (and thus slightly rearrange the sequence map expression to maintain bindings for $v_1$). Outside of the sequence map expression standard relational algebra is used to perform the additional join:

$$
\begin{aligned}
\text{BINDINGS} &:= \pi_{v_3}(\rho_{1\to v_1,2\to v_3}(\text{path}_{1,1}) \bowtie F_{v_1,v_3}(\ddot{\bar{\omega}}^{v_1}_{\downarrow}(\ddot{\bar{\omega}}^{v_2}_{\downarrow}(\ddot{\bar{\omega}}^{\uparrow}_{v_3}(\ddot{\bar{\omega}}^{\uparrow}_{v_2}( \\
&\quad \ddot{\mu}_{v_1}(D, Q, \text{root}) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_1}(D, Q, \text{'conference'}) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_1,v_2}(D, Q) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2}(D, Q, \text{'paper'}) \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_2,v_3}(D, Q) \\
&\quad \ddot{\bowtie}^{\lightning} \ddot{\mu}_{v_3}(D, Q, \text{'author'})))))))
\end{aligned}
$$

Translating the head is a bit more involved. The reason lies, on the one hand, with the order and node invention, but even more with the deep-copy construct. First, recall (from Section 7.4) that the head

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1, \texttt{authors}) \wedge \textsc{child}(\mathsf{id}_1, \mathsf{id}_2(v_3)) \wedge \mathsf{deep\text{-}copy}(v_3, \mathsf{id}_2(v_3))$$

is actually an abbreviation for

$$\texttt{root}(\mathsf{id}_1) \wedge \mathfrak{L}(\mathsf{id}_1, \texttt{authors}) \wedge \circ\!\!\rightarrow(\mathsf{id}_1, \mathsf{id}_3(v_3)) \wedge \rightarrow\!\!\circ(\mathsf{id}_2(v_3), \mathsf{id}_3(v_3)) \wedge \mathsf{pos}(\mathsf{id}_3(v_3), \mathsf{id}_4(v_3)) \wedge \mathsf{deep\text{-}copy}(v_3, \mathsf{id}_2(v_3))$$

It is also worth pointing out that, as discussed in Chapter 6, we do not construct all CIQLog relations but only those that can not be derived from others, i.e., the extensional relations *source*, $\rightarrow\!\!\circ$, pos, root, $\mathfrak{L}$, $\mathfrak{O}$.

When translating a head, we first introduce a new relation for each unique value invention or order invention term. The relation extends the binding tuples constructed by the expression corresponding to the CIQLog body with appropriate new values or order numbers. Here we have four such relations (we choose to name the last one differently to highlight that it contains order numbers):

$$\begin{aligned}
\textsc{id}_1 &:= \iota_{new1:\langle\rangle}(\textsc{bindings}) \\
\textsc{id}_2 &:= \iota_{new2:\langle v_3\rangle}(\textsc{bindings}) \\
\textsc{id}_3 &:= \iota_{new3:\langle v_3\rangle}(\textsc{bindings}) \\
\textsc{order}_3 &:= \iota_{order:\langle v_3\rangle}(\textsc{bindings})
\end{aligned}$$

Second, for each deep-copy we have to determine the reachable nodes and edges between those and for each such node and edge create corresponding new nodes or edges:

$$\begin{aligned}
\textsc{reachable-nodes}_{v_3} &:= \pi_{v_3}(\textsc{bindings}) \bowtie \rho_{1\to v_3, 2\to old}(\mathsf{path}_*) \\
\textsc{reachable-edges}_{v_3} &:= \pi_{v_3, oldEdg}(\textsc{reachable-nodes}_{v_3} \\
&\qquad \bowtie \rho_{1\to old, 2\to oldEdg}(\circ\!\!\rightarrow)) \\
&\quad \cap \pi_{v_3, oldEdg}(\textsc{reachable-nodes}_{v_3} \\
&\qquad \bowtie \rho_{1\to old, 2\to oldEdg}(\rightarrow\!\!\circ)) \\
\textsc{copied-nodes}_{v_3} &:= \iota_{new:\langle v_3, old\rangle}(\textsc{reachable-nodes}_{v_3}) \\
\textsc{copied-edges}_{v_3} &:= \iota_{newEdg:\langle v_3, oldEdg\rangle}(\textsc{reachable-edges}_{v_3})
\end{aligned}$$

Notice, how we record both the grouping value $v_3$ and the original nodes or edges. This allows us to copy the correct properties by adding the new nodes (edges) to all relations that also contained the old nodes (edges):

$$
\begin{aligned}
\textsc{root-copy} &:= \pi_{new}\big(\rho_{1\to old}(\mathsf{root}) \bowtie \textsc{copied-nodes}_{v_3}\big)\\
\textsc{source-copy} &:= \pi_{new,newEdg}\big(\rho_{1\to old,2\to oldEdg}(\circ\!\!\to) \bowtie \textsc{copied-nodes}_{v_3}\\
&\qquad \bowtie \textsc{copied-edges}_{v_3}\\
&\ldots
\end{aligned}
$$

We do this for all base relations but show only the first few. The remaining base relations are treated analogously.

The final result of the CIQLog rule are then new assignments for the base relations that are constructed by a union of the relations just created and whatever is specified directly in the rule head. Here, e.g., we create additional entries in root, $\mathfrak{L}$, $\circ\!\!\to$, $\to\!\!\circ$, and pos as these occur in the rule head.

$$
\begin{aligned}
\mathsf{root} &:= \pi_{new}(\textsc{id}_1) \cup \textsc{root-copy}\\
\mathfrak{L} &:= \pi_{new1}(\textsc{id}_1) \times \{\langle \text{'authors'} \rangle\} \cup \textsc{label-copy}\\
\circ\!\!\to &:= \pi_{new1,new3}(\textsc{id}_1 \bowtie \textsc{id}_3) \cup \textsc{source-copy}\\
\to\!\!\circ &:= \pi_{new2,new3}(\textsc{id}_2 \bowtie \textsc{id}_3) \cup \textsc{sink-copy}\\
\mathsf{pos} &:= \pi_{new3,order}(\textsc{id}_3 \bowtie \textsc{order}_3) \cup \textsc{position-copy}\\
\mathfrak{O} &:= \textsc{ordered-copy}
\end{aligned}
$$

## 13.3 CIQLOG TRANSLATION

Formally, we specify the translation function below. The first step of the translation is to determine a spanning tree over the body of the CIQLog query. To allow for efficient computation of the spanning tree, we assume a highly simplified cost function that assigns weights directly to unary and binary relations in the query regardless of their position relative to other relations. Such a cost function can only be a very rough approximation of the actual cost, but allows us to use any of the efficient spanning tree algorithms, cf. [6].

If we use cost functions where the cost of operators depends on their relative position, these algorithms can no longer be applied and the problem of determining an optimal spanning tree becomes NP-complete (like other constrained spanning tree problems, e.g., the Optimum Communication Spanning Tree problem).

Finding an efficient heuristic that yields nearly optimal spanning trees under such cost functions remains an open problem.

### 13.3.1 TRANSLATION FUNCTION

Given a conjunction of CIQLOG literals, we assume that we can compute a tree core *tree-core* for that conjunction and denote the variables in *tree-core* $\mathcal{V}_T$, all other variables $\mathcal{V}_G$.

First, we transform each CIQLOG rule into the form

$$head \longleftarrow b_1 \vee b_2 \vee \ldots \vee b_k$$

where each $b_i$ is a conjunction of the form $\mathit{tree\text{-}core}(b_i) \wedge \ldots$. The body of such rules is translated by the translation function tbody given in Table 67.

| function | CIQLOG expression | CIQCAG expression |
|---|---|---|
| tbody$\langle \mathit{rel}(\mathit{var}_1, \mathit{var}_2) \rangle$ | | $= \rho_{1 \to \mathit{var}_1, 2 \to \mathit{var}_2}(\mathit{rel})$ |
| tbody$\langle \mathit{rel}(\mathit{var}) \rangle$ | | $= \rho_{1 \to \mathit{var}}(\mathit{rel})$ |
| tbody$\langle \mathit{expr}_1 \vee \mathit{expr}_2 \rangle$ | | $= \pi_{\mathcal{V}_G}(\text{tbody}\langle \mathit{expr}_1 \rangle) \cup \pi_{\mathcal{V}_G}(\text{tbody}\langle \mathit{expr}_2 \rangle)$ |
| tbody$\langle \mathit{tree\text{-}core} \wedge \mathit{expr} \rangle$ | | $= F_{\mathcal{V}_{E_1}}(\ddot{\omega}^{v_1}_{\downarrow}(\ldots \ddot{\omega}^{v_k}_{\downarrow}(\ddot{\omega}_{v_k^{\uparrow}}(\ldots(\ddot{\omega}_{v_1^{\uparrow}}(S))\ldots))\ldots)) \bowtie \pi_{\mathcal{V}_{E_2}}(\text{tbody}\langle \mathit{expr} \rangle)$<br>**where** $\mathcal{V}_{E_1} = \text{Vars}(\mathit{tree\text{-}core}) \cap (\text{Vars}(\mathit{expr}) \cup \mathcal{V}_G)$<br>$\mathcal{V}_{E_2} = \text{Vars}(\mathit{expr}) \cap (\text{Vars}(\mathit{tree\text{-}core}) \cup \mathcal{V}_G)$<br>$S = \text{ttree}\langle \mathit{tree\text{-}core} \rangle$<br>$v_1, \ldots, v_k$ variables of $S$ in topological order induced by the edge cover of $S$ |
| tbody$\langle \mathit{expr}_1 \wedge \mathit{expr}_2 \rangle$ | | $= \pi_{\mathcal{V}_{E_1}}(\text{tbody}\langle \mathit{expr}_1 \rangle) \bowtie \pi_{\mathcal{V}_{E_2}}(\text{tbody}\langle \mathit{expr}_2 \rangle)$<br>**where** $\mathcal{V}_{E_1} = \text{Vars}(\mathit{expr}_1) \cap (\text{Vars}(\mathit{expr}_2) \cup \mathcal{V}_G)$<br>$\mathcal{V}_{E_2} = \text{Vars}(\mathit{expr}_2) \cap (\text{Vars}(\mathit{expr}_1) \cup \mathcal{V}_G)$ |
| tbody$\langle \mathit{expr}_1 \wedge \neg(\mathit{expr}_2) \rangle$ | | $= \text{tbody}\langle \mathit{expr}_1 \rangle \setminus \text{tbody}\langle \mathit{expr}_1 \wedge \mathit{expr}_2 \rangle$ |

Table 67. Translating CIQLOG rule bodies

Most notably, for the translation of the tree core(s) a second translation function ttree detailed in Table 69 is used. It employs sequence map operators where tbody uses standard relational operators. It also uses a helper function $\text{lca}_Q(V)$ with $V$ set of variables which returns the set of all least common ancestors of any pair of variables from $V$ in the query $Q$.

Finally, the head of a CIQLOG rule is translated by thead as illustrated in the above example: **(1)** for each node and order invention term a corresponding relation using a invention or numbering operator to invent the new values is introduced **(2)** for each deep-copy all reachable nodes and edges between

| function | CIQLOG expression | CIQCAG expression |
|---|---|---|
| | $\mathrm{ttree}\langle rel(var_1, var_2)\rangle$ | $= \ddot{\mu}_{var_1, var_2}(D, Q)$ |
| | $\mathrm{ttree}\langle rel(var)\rangle$ | $= \ddot{\mu}_{var}(D, Q, rel)$ |
| | $\mathrm{ttree}\langle rel_1(var) \vee rel_2(var)\rangle$ | $= \mathrm{ttree}\langle rel_1(var)\rangle \ddot{\cup} \mathrm{ttree}\langle rel_2(var)\rangle$ |
| | $\mathrm{ttree}\langle rel_1(var_1, var_2) \vee rel_2(var_1, var_2)\rangle$ | $= \mathrm{ttree}\langle rel_1(var_1, var_2)\rangle \ddot{\cup} \mathrm{ttree}\langle rel_2(var_1, var_2)\rangle$ |
| | $\mathrm{ttree}\langle expr_1 \wedge expr_2\rangle$ | $= \ddot{\pi}_{\mathcal{V}_{E_1}}(\mathrm{ttree}\langle expr_1\rangle)\ \theta\ \ddot{\pi}_{\mathcal{V}_{E_2}}(\mathrm{ttree}\langle expr_2\rangle)$ **where** $\mathcal{V}_{E_1} = \mathcal{V}'_{E_1} \cup \mathrm{lca}_Q(\mathcal{V}'_{E_1})$ with $\mathcal{V}'_{E_1} = \mathrm{Vars}(expr_1) \cap (\mathrm{Vars}(expr_2) \cup \mathcal{V}_G)$ $\mathcal{V}_{E_2} = \mathcal{V}'_{E_2} \cup \mathrm{lca}_Q(\mathcal{V}'_{E_2})$ with $\mathcal{V}'_{E_2} = \mathrm{Vars}(expr_2) \cap (\mathrm{Vars}(expr_1) \cup \mathcal{V}_G)$ $\theta = \begin{cases} \ddot{\bowtie}^{\not\downarrow} & \text{if } \mathcal{V}_{E_1} \subset \mathcal{V}_{E_2} \\ \ddot{\bowtie}^{\not\downarrow} & \text{if } \mathcal{V}_{E_2} \subset \mathcal{V}_{E_1} \\ \ddot{\bowtie}^{\not\downarrow} & \text{otherwise} \end{cases}$ |

Table 69. Translating CIQLOG tree cores

them are copied and the properties of the original nodes transferred. **(3)** for each other atom in the head, we add a corresponding expression to the union representing the atom's base relation.

**Theorem 13.1.** *The result of the above translation from* CIQLOG *to* CIQCAG *is a* CIQCAG *expression that implements the semantics of the* CIQLOG *rule and is linear in the size of the* CIQLOG *rule.*

*Proof.* For the most part the algebraic semantics of CIQLOG in Table 5 is closely reflected in the translation function. For the translation of the tree core, see Chapter 12 for proofs that the sequence map operators have the same effect on the induced relation of the sequence map as the standard relational operators.

For the translation of the head, the crucial issues are, once again, the handling of order and node invention. The corresponding operators are specifically aligned with the semantics of order and node invention as discussed in Chapter 6.

It is easy to recognize that the resulting expression is *linear*, if one realizes that we use assignment expressions to avoid repeating sub-expressions that occur in multiple places (e.g., the BINDINGS relation that occurs in each node invention relation and thus, if naïvely repeated each time results

in a quadratic size for the resulting expression. Note, that the translation of deep-equal introduces a rather large, but constant number of CIQCAG expressions. □

## 13.4 ITERATION AND RECURSION

Following Abiteboul [2], we complete the CIQCAG algebra (so as to support full CIQLog) by an iteration operator **while**. For simplicity, we assume an inflationary semantics for the assignment operator in a **while**. To translate a general CIQLog program (with inflationary semantics), we enclose the relation assignments generated by the above rule in **while** loops that terminate if there are no more changes. Obviously, termination is not guaranteed in contrast to datalog as recursion over value invention terms (cascading value invention) may create an unbounded, possibly infinite number of new values.

Combining the CIQCAG algebra with alternative semantics for recursion is a topic of ongoing investigations and remains, for this work, an open question.

## 13.5 CONCLUSION

The translation from CIQLog to CIQCAG concludes our journey that starts in Chapter 5 where we ask how to formalize Web queries. With CIQLog we have a convenient, logical foundation of Web query languages as diverse as XQuery, Xcerpt, XPath, and SPARQL (see Part III) that is complemented by an efficient, scalable query algebra, CIQCAG, that realizes tree queries even on many graphs in linear time and space, surpassing previous approaches significantly. In this chapter, we conclude the consideration of CIQCAG by (very briefly) outlining how to realize non-tree parts of a query as well as construction in standard relational algebra with value and order invention operators (provided by most practical SQL database systems).

To complete the picture, we turn in the remaining chapter to the implementation of CIQCAG and evaluate the discussed approach experimentally.

Part V

# PRACTICE. THE CIQCAG PROTOTYPE

# 14 PROTOTYPE AND EXPERIMENTAL EVALUATION



## 14.1 INTRODUCTION

In Part IV we introduce the CIQCAG algebra and discuss its properties. In this chapter, we *experimentally verify* these properties, most notably the following three properties: **(1)** Tree queries can be evaluated by the CIQCAG algebra in linear time and space on both tree data and continuous-image graphs (CIGS), i.e., graphs with some order on the nodes such that the children of each node form a continuous interval. **(2)** Graph queries can be evaluated in two phases, first some spanning tree of the query, then the remaining non-tree relations. Only answer nodes and nodes with non-tree edges are considered also in the graph phase. Thus, only these nodes contribute to the exponent of the complexity $\mathcal{O}(n^{q_g})$ where $n$ is the size of the data and $q_g$ the number of answer nodes and nodes with non-tree edges. **(3)** Furthermore, we introduce the sequence map data structure and argue that even for many non-CIG graphs it provides a more efficient (interval) representation than standard relational algebra. It is an open issue how to efficiently find the optimal order of the nodes of a graph to obtain a minimal interval representation. However, in practical graphs we observe often a strong hierarchical component with few non-hierarchical outliers. In such cases the order obtained for the hierarchical component

forms a good starting point as basis for compact interval representation of the whole graph.

We start with a brief discussion of the CIQCAG prototype and then present a number of experiments that investigate the above properties. The experiments fully support the theoretical results from previous chapters and demonstrate that, even with the currently rather basic prototype, query evaluation with CIQCAG performs exceptionally well for practical queries and data with response times for 15-20 variable queries on 25-50 MB data sets well below 1 sec.

## 14.2 CIQCAG PROTOTYPE

The CIQCAG prototype is still under development. A preliminary version has been used to perform the tests described below. The current version is a rather preliminary version. It only implements the core elements of the CIQCAG approach, as depicted in Figure 66.

**(1)** The *translation* of large parts of the *Web query languages* XPath, XQuery, Xcerpt, and SPARQL, as described in Part III, into CIQLog queries forms the first part of the prototype. It consider mostly the composition-free or non-recursive fragment of these languages and are, consciously, limited to the language core described in Part III. The result are CIQLog queries.

**(2)** The CIQCAG *compiler* translates CIQLog queries to CIQCAG expressions. The current implementation is rather basic and computes an arbitrary spanning tree to separate tree core from non-tree relations (unless the query is already tree shaped). The result of the algebra compiler are CIQCAG expressions.

**(3)** The *tree query processor* is based on a straightforward implementation of the sequence map and its operators as described in Chapters 11 and 12. Currently only the sequence map operators generated by the algebra compiler are supported. This excludes, e.g., union, difference, and rename. Though the algebra compiler can generate either the consistent or the inconsistent variants of the sequence map operators, the tree query processor currently only implements the latter. Recall from Section 12.1, that allowing inconsistent sequence maps as result of an operator yields, in many cases, more efficient (and simpler) realizations of the various operators. The price is the need for additional propagation operators, see Section 12.5.3 that remove any inconsistencies before extracting the correct answers

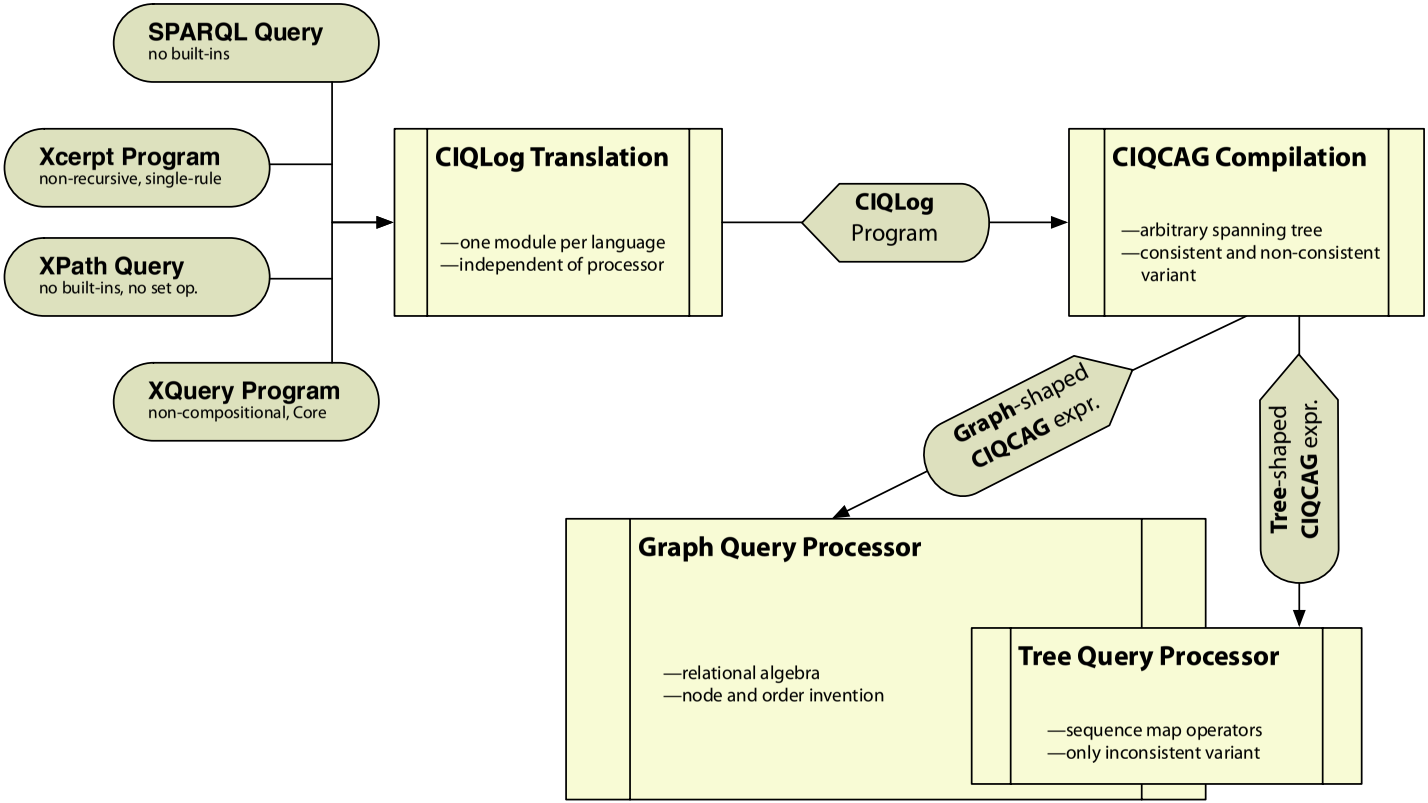



Figure 66. CIQCAG Prototype: Overview

from the sequence map. The tree query processor can be used separately from the graph query processor (if the entire query is tree shaped).

**(4)** The *graph query processor* implements the full CIQCAG algebra. For sequence map operations it calls the tree query processor. Non-tree relations, node, and order invention are implemented directly in the graph query processor. Both the graph and the tree processor currently have only very crude data access layers: both expect relations in interval representation as input without attempting to find more optimal representations. This is considered to be performed by a separate data access or indexing component that pre-computes optimal (or at least almost optimal) interval representations of given data and stores the data in that representation. However, the current prototype does not include such a component, but leaves that task to the user.

It is worth pointing out that the CIQLOG translation and the CIQCAG compiler are independent components with well-defined interfaces (CIQLOG queries and CIQCAG expressions, resp.). In fact, in the current prototype they are not even implemented in the same programming language: The core processors are implemented in C++, whereas the translation and compiler are a combination of ANTLR[1] and Java. The current prototype is based on the one used in [52].

## 14.3 EXPERIMENTAL EVALUATION

SETUP. The experimental evaluation is based on both synthetic and on real data. The data is assumed in interval representation with the relations discussed in Chapter II. All tests show the processing time without data parsing. The tests have been executed on an AMD Athlon 2400XP CPU with 1GB main memory. The implementation currently uses only a single core. Each measurement is averaged over 500 runs. For the tests, we have used queries manually written in CIQCAG rather than the output of the compiler. However, the queries are close to what the compiler generates for basic XPath, Xcerpt, and XQuery queries. We have not yet studied the effect of the frequent label joins in the SPARQL translation.

Synthetic data is used to confirm the complexity of the presented algorithms. The real data scenarios stem from the University of Washington

1 http://www.antlr.org/

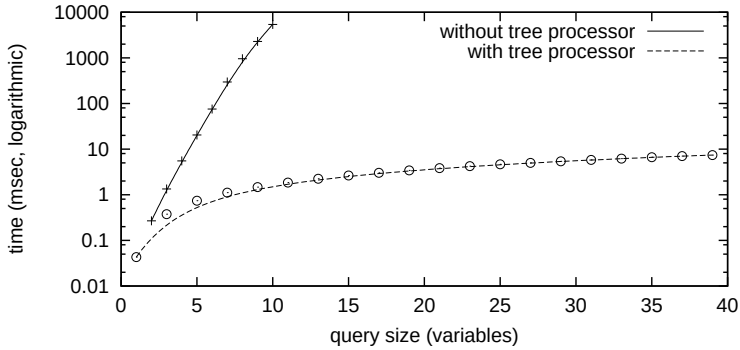



Figure 67. Effect of Sequence Map Operators (over query size) (data synthetic, uniform, deeply nested; bindings for query variables overlap considerably)

XMLData repository[2], and demonstrate the competitiveness of the algorithms.

### 14.3.1 EFFECT OF SEQUENCE MAP

First, we consider the effect of using specialized operators for the evaluation of tree queries. For that, we compare the evaluation time of the same query, once realized with standard relational operators and evaluated in the graph query processor, once realized with sequence map operations and evaluated in the tree query processor. As expected, time (see Figure 67) and space (not shown) usage of the evaluation are roughly linear in the number of variables for the second case, but exponential for the first case. This validates the practical soundness of using an algebra with specialized tree query operators.

For the experiment we use uniform, deeply nested XML data and a fairly unselective query composed of successive descendant-or-self ($\text{path}_*$) with a single answer variable. The latter suppresses the influence of answer variables on the result as discussed in the following. It is worth pointing out that, in and by itself, this is not a surprising result as similar observations for acyclic or tree queries [110] that can, e.g., be realized with a semi-join operator [146] are well-established. However, here we consider a rather rich algebra of sequence map operators, including forms of union and difference. Furthermore, the interval representation allows for linear space representations even of $n$-ary queries where the semi-join algebra and

2 http://www.cs.washington.edu/research/xmldatasets/

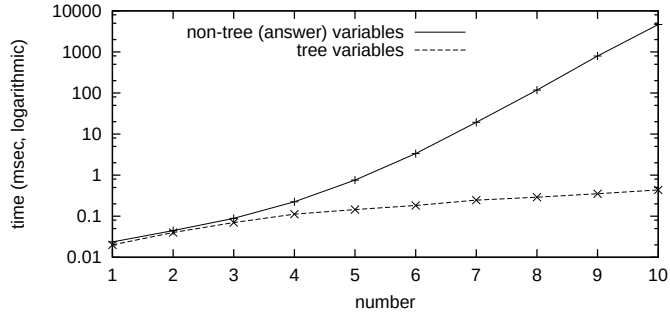



Figure 68. Effect of Tree vs. Non-Tree Variables (data and query as before)

similar approaches require at least quadratic space.

### 14.3.2 EFFECT OF NON-TREE EDGES

In contrast to the previous experiment, increasing the number of answer variables or variables with incident non-tree edges considerably affects the evaluation time. Figure 68 compares the effect of increasing the two types of variables, those occurring also in non-tree relations or in the answer with those occurring only in tree relations and not in the answer. The latter have, as in the previous experiment, little effect on the processing time. The former, however, significantly increase query processing time, thus emphasizing the complexity result for graph queries from Chapter 13: $\mathcal{O}(n^{q_g} \cdot n \cdot q_t)$, where $q_g$ is the number of variables in the answer or in non-tree relations and $q_t$ the number of variables that occur only in tree relations but not in the answer.

Again the experiment uses synthetic queries and data. The data is rather small and the selectivity, as in the previous case, is not significantly affected by the number of variables. Thus, this is certainly an extreme setting for studying the effect of non-tree variables. In most practical cases, it can, e.g., be expected that not all bindings of one answer variable are related to those of all bindings of another as in this case.

### 14.3.3 EFFECT OF DATA SHAPE

For the final two experiments, we use real-life XML datasets from the above mentioned repository. In the first experiment, we illustrate the effect of the data shape on the query evaluation time. To compare tree, CIG, and graph data we start with the tree-shaped Nasa dataset from above. To that data, we add non-tree edges in such a way that the result is still CIG data.

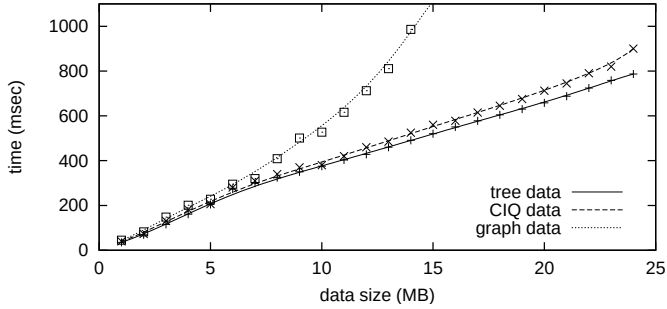



Figure 69. Query Evaluation Time over Different *Data* Shapes (tree query)

The number of added edges is such that each node has on average between 2 and 3 parents, i.e., the resulting data contains about 2.5 times the number of edges as the original tree data. Finally, we add to the resulting data set arbitrary additional non-tree edges, but also delete some edges to obtain the graph data set. We delete edges to create non-continuous intervals. The resulting data has about 3 times the number of edges as the original tree data. The interval representation of the graph data is over the same order on the nodes as for the CIG data set (and thus not necessarily optimal). It requires multiple intervals to represent the images of most nodes, on average about 3 intervals per node. Finally, the used query is tree-shaped, contains about 15 variables, and matches with about 5% of the data.

Under this setup, Figure 69 shows that, indeed, the performance of tree query evaluation is nearly the same on CIG data as on tree data. Furthermore, it is clearly linear and is very competitive, even though the current prototype is only a very basic implementation. For graph data, as expected, the query evaluation time increases faster, but is still rather acceptable with about 1 sec for a 10 MB fragment of the Nasa dataset.

### 14.3.4 EFFECT OF QUERY SHAPE.

The final experiment considers, again, the effect of the query shape, but now on a real-life data set, viz. the MONDIAL[3] database of geographical information and with realistic queries. The data is tree shaped. In particular, the graph query contains only a small (about 20%) amount of non-tree variables. The queries are otherwise fairly involved compared to the above examples but are also far more selective than the queries of the other experiments. The results in Figure 70 again emphasize the linear time

3 http://www.dbis.informatik.uni-goettingen.de/Mondial/

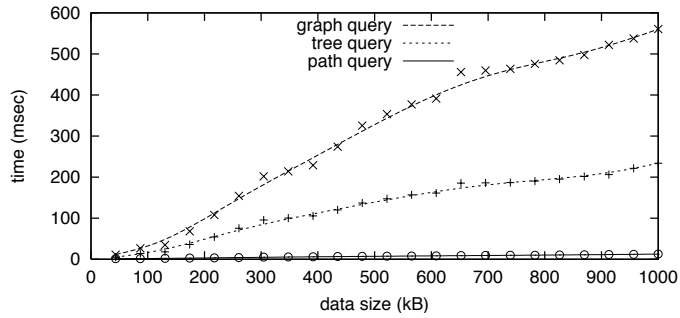



Figure 70. Query Evaluation Time over Different *Query* Shapes (Mondial dataset)

processing for tree queries and show that practical graph queries can often also be evaluated efficiently by the proposed approach.

## 14.4 OUTLOOK: PRINCIPLES OF THE CIQCAG PROCESSOR

The current CIQCAG prototype is fairly basic, but will be further towards a flexible, distributed evaluation engine for Web query languages. The basic architecture and principles for that next-generation CIQCAG processor are described in [50]. In the following, we briefly summarize the most important points of that architecture to highlight the planned future development of the CIQCAG processor:

"EXECUTE ANYWHERE"—UNIFIED QUERY EXECUTION ENVIRONMENT. CIQLOG and the CIQCAG algebra provides a unified execution environment for Web queries written in a multitude of query languages (as demonstrated in Part III. In this respect, CIQCAG can be considered a similar foundation for Web query languages as multi-language virtual or abstract machines for standard programming languages. The most common examples of multi-language virtual machines is the Common Language Infrastructure [134] used prominently for executing C♯, but also the Java Virtual Machine is increasingly used as target for a wide variety of languages.

In the case of Web queries, a unified execution environment brings a number of unique advantages: **(1)** The *distributed execution of queries and query programs* requires that the language implementations are highly interoperable down to the level of answer representation and execution strategies. A high degree of interoperability allows, e.g., the distribution of partial queries among query nodes (see below). The CIQCAG algebra is

an a suitable mechanism to ensure implementation interoperability as its operations are fairly fine granular and well-specified allowing the controlling query node fine granular control over the query execution at other ("slave") nodes. **(2)** A rigid definition of the operational semantics as provided by an the CIQCAG algebra allows not only a better understanding and communication of the evaluation algorithms, it also makes *query execution more predictable*, i.e., once compiled a query should behave in a predictable behavior on all implementations. This is an increasingly important property as it eases query authoring and allows better error handling for distributed query evaluation. **(3)** Finally, a unified query execution environment makes the *transmission and distribution of compiled queries and even parts* of compiled queries among query nodes feasible, enabling easy adaptation to changes in the network of available query nodes, see below.

"COMPILE ONCE"—SEPARATION OF COMPILATION AND EXECUTION.

Currently, the CIQCAG processor targets the in-memory processing of queries against XML, RDF, or other Web data that may be local and persistent (e.g., an XML database or local XML documents), but just as well may have to be accessed remotely (e.g., a remote XML document) or may be volatile (e.g., in case of SOAP messages or Web Service access). In other words, it is assumed that most of the queried data is not under (central) control of a query execution environment like in a traditional database setting, but rather that the queried data is often distributed or volatile. This, naturally, limits the application of traditional indexing and predictive optimization techniques, that rely on local management of data and statistic knowledge about that managed data.

Nevertheless *algebraic optimization techniques* (that rely solely on knowledge about the query and possibly the schema of the data, but not on knowledge about the actual instance of data to be queried) and *ad-hoc indices* that are created during execution time still have their place under this circumstances.

In particular, such a setting allows for a clean *separation of compilation and execution*: The high-level Web queries in XQuery, XPath, Xcerpt, or SPARQL are translated into CIQCAG expressions separately from its execution. The translation may be separated by time (at another time) and space (at another query node) from the actual execution of the query. This is essential to enable the distribution of pre-compiled, globally optimized CIQCAG expressions evaluating (parts of) queries over distributed query nodes.

*Extensive static optimization.* This separation also makes more extensive

static optimization feasible than traditionally applied in an in-memory setting (e.g., in XSLT processors such as Saxon[4] or Xalan[5]).

"COMPILE, CLASSIFY, EXECUTE"—UNIFIED EVALUATION ALGORITHM. Aside of traditional tasks such as dead (or tautological) branch elimination, detection of unsatisfiable queries, operator order optimization and selection between different realizations for the same high-level query constructs, the CIQCAG compiler has another essential task: the *classification of each query* in the query program by its features, e.g., whether a query is a path, tree, or graph query (cf. [170, 52]) or which parts of the data are relevant for the query evaluation.

"DISTRIBUTE ANY PART"—PARTIAL QUERY EVALUATION. Once compilation and execution are separate, the possibility exists that one query node compiles the high-level Web queries to CIQCAG code using knowledge about the query and possibly the schema of the data to optimize (globally) the query plan expressed in the CIQCAG code. The result of this translation can then be distributed among several query nodes, e.g., if these nodes have more efficient means to access the resources involved in the query.

Indeed, once at the level of CIQCAG code it is not only possible to distribute, say, entire CIQLOG rules or sets of CIQLOG rules, but even parts of rules (e.g., query conjuncts) or even smaller units. Figure 71 illustrates such a distributed query processing scenario using Xcerpt as high-level language (though any other language, and with some care, also a mixture of languages is possible): Applications use one of the control APIs (obtaining, e.g., entire XML documents or separate variable bindings) to execute a query at a given Xcerpt node. This implementation of Xcerpt transforms the query into CIQCAG code and hands this code over to its own CIQCAG engine. Depending on additional information about the data accessed in the query, this CIQCAG node might decide to evaluate only some parts of the query locally (e.g., those operating exclusively on local data and those joining data from different sources) and send all the remaining query parts to other CIQCAG nodes that are likely to have more efficient access to the relevant data.

In contrast to distribution on the level of a high-level query language such as Xcerpt, distribution on the level of CIQCAG has two main advantages: the distributed query parts can be of finer granularity and the "controlling" node can have, by means of code transformation and hint sections, better

4 http://www.saxonica.com/

5 http://xml.apache.org/xalan-j/

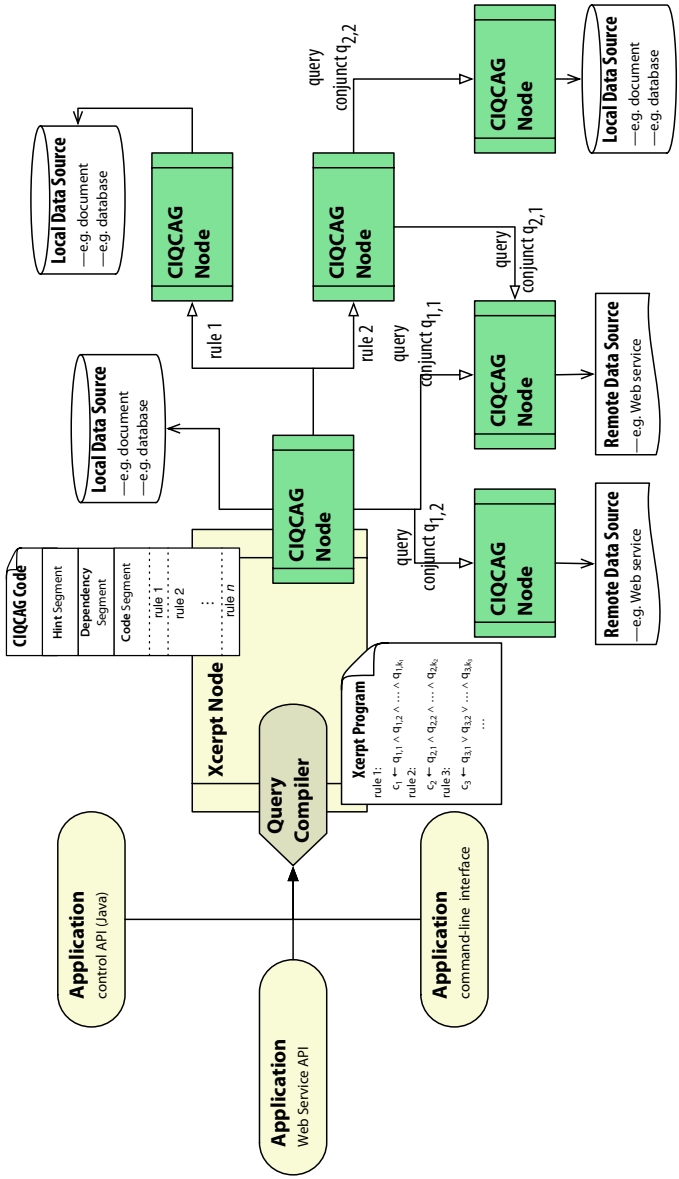



Figure 71. Query Node Network

control of the "slave" nodes.

Notice, that CIQCAG enables such query distribution, but does not by itself provide the necessary infrastructure (e.g., for registration and management of query nodes). It is assumed that this infrastructure is provided by outside means.

## 14.5 CONCLUSION

Though the current prototype is only a fairly preliminary implementation of the CIQCAG algebra, the above experiments already clearly demonstrate the viability of the approach. Employing interval representations of relations and specialized sequence map operators for tree query evaluation, the CIQCAG prototype achieves:

**(1)** Linear time and space evaluation for tree queries on tree and CIG data. In fact, the experiments show that there is no significant difference in evaluation time between tree and CIG data. In other words, non-tree edges come for free w.r.t. query evaluation as long as the resulting graph still carries the CIG property.

**(2)** Even for arbitrary graph data interval representations yield an efficient basis for evaluation since they provide, except for extreme cases, compact representations of related nodes at little extra cost.

**(3)** The separate processing of tree core for graph queries yields for many practical graph queries a significant improvement, in particular as most graph queries have only limited non-tree edges.

**(4)** All major complexity results from the previous chapters are validated by the above experimental evaluation.

It is expected that work on the prototype will continue in the future. In particular, a full implementation of the CIQCAG algebra, more extensive experimental evaluations, and the realization of the architecture described in the previous chapter are topics for future work.

# 15 CONCLUSION



In three parts, this thesis illustrates perspectives to overcome the increasing fragmentation of the Web in HTML, XML, RDF, OWL, etc. islands. The fundamental results are:

**(1)** A *versatile query language*—that can access data in many Web formats and in many shapes—is viable as demonstrated by Xcerpt 2.0 in Chapter 3. Furthermore, it does not need to sacrifice ease of use for each of the formats as illustrated along the GRDDL use case in Chapter 4.

**(2)** Though the fragmentation of the Web has also introduced a plethora of query languages, each specialized to one data format, a *uniform, purely logical semantics for* such widely varying languages as *XPath, XQuery, SPARQL, and Xcerpt* is provided in Part II and III. For this purpose, we introduce CIQLog as a slightly modified variant of $\text{datalog}^{\neg}_{new}$, i.e., datalog extended with negation and value invention. We show that all the above languages can be faithfully translated into CIQLog.

As a side effect, we obtain the first, to the best of our knowledge, purely logical semantics of XQuery and SPARQL and illustrate commonalities and differences between these languages. Furthermore, the uniform semantics allows us to expect that, in many cases, integration of queries in these different languages is possible.

**(3)** The uniform semantics for Web query languages provides by CIQLog is complemented in Part IV by the CIQCAG algebra used for implementing CIQLog queries. CIQCAG is the first algebra for Web queries that scales with query and data shape: For tree queries it provides linear time and space evaluation (cf. Section 11.4 and Chapter 12) on tree data and thus the same complexity as the best known approaches for that setting. Furthermore, it extends the same complexity also to a far larger, novel class of graphs, called continuous-image graphs (Sections 10.2 and 11.3) that can still be processed with linear time and space. In this respect the move from trees to continuous-image graphs comes for free w.r.t. query evaluation time. But also on arbitrary graph data, the CIQCAG algebra can provide significant speed-up (as experimentally verified in Chapter 14) compared with prior approaches, in particular if the graph is removed from a tree or CIG only by a few edges.

For graph queries, CIQCAG evaluates a tree core of the query first (with, for tree and CIG data, linear or, for arbitrary graphs, at worst quadratic time and space complexity). Only those parts of the query with relations not covered by the tree core are then processed using standard relational operators, thus reducing the complexity of query evaluation to size of those non-tree parts of a query (see Chapter 13). Combined with the exceptionally efficient evaluation for tree queries discussed above, this yields an overall highly competitive engine for evaluating Web queries.

With the results on continuous-image graphs, the CIQCAG algebra extends the limits of previously known approaches with linear time and space query evaluation considerably into the realm of graph-shaped data. This not only makes versatile query languages and multi-language engines for Web queries more feasible, it also gives an indication how some of the techniques developed for XML and similar, mostly tree-shaped, data formats can be extended or adapted for use with less restricted, graph-shaped data such as RDF.

## 15.1 PERSPECTIVES AND FURTHER WORK

Obviously, this work provides only a first few attempts at addressing the increasing fragmentation of the Web into islands of diverging data formats. In the following, we briefly highlight a few venues for further work.

### 15.1.1 CONTINUOUS-IMAGE GRAPHS

As mentioned in Chapter 10, there are a number of questions that have arisen in our investigation of continuous-image graphs as a class of data where tree queries can be evaluated in linear time and space:

**(1)** Though we have illustrated the use of CIGs by a number of examples, a principled investigation of the *frequency of CIGs in practical data sets* is certainly desirable. In general, studies on the precise characteristics of Web or Semantic Web data are few and far between. Issues such as the average and maximum degree of concepts in Ontologies or their overall organization (e.g., around hierarchical components) can significantly affect the performance of querying and reasoning with such data, yet are often ignored. This work illustrates that a more precise characterisation of Web data (beyond tree- vs. graph-shaped) can be helpful in addressing some of the scalability concerns raised in the context of the Semantic Web.

In particular, further investigating the amount and significance of non-tree (or non-CIG) edges would allow a better understanding when interval representations of arbitrary graphs are useful to speed up query evaluation.

**(2)** In this work, the nodes in the image of each node of a CIG are required to form a *single* continuous interval. For many graphs, no order on the nodes may exists such that there is a single such interval, but only orders such that there are $k$ such intervals where $k$ is some small integer. Such graphs are then still amenable to linear time and space evaluation with sequence maps. However as indicated in Section 11.3.3, it is an open question whether there is a polynomial decision algorithm for the *$k$-image interval property*. Though the test for "circular ones" points towards such an algorithm for $k = 2$, there does not seem to be such an obvious extension of the decision algorithm for $k = 1$ to cases with $k > 2$.

**(3)** In Section 10.3.2, we briefly discuss that the sequence map can also handle *diamond-free DAG queries* while retaining the same data complexity. The query complexity only remains the same, for obvious reasons, if we measure the size of a query as the number of *edges* (i.e., atoms in a CIQLOG expression) rather than the number of *nodes* (i.e., variables in a CIQLOG expression). However, a detailed consideration of diamond-free DAG queries remains open, yet might yield interesting results as there is some evidence [170, 167] that such queries occur frequently, at least in the XPath context.

**(4)** Similarly, we have limited ourselves in this work to tree queries though there are larger classes of queries with polynomial query evaluation time, viz. queries with *bounded hypertree width* [109]. As discussed in Section 10.4.2, it is not obvious whether and how the notion of CIGs can be combined with such queries. A straightforward adaptation by extending the CIG property to guarantee one order for all relations in a hypertree node seems considerably more restrictive than the basic notion of a CIG graph. Whether there are more promising ways to adapt the sequence map data structure and the notion of CIGs for queries with bounded hypertree width remains an open question.

### 15.1.2 ITERATOR IMPLEMENTATION OF THE SEQUENCE MAP

The sequence map is the means in CIQCAG to provide linear time and space evaluation of tree queries on tree and CIG data. In Section 12.9, we discuss how an iterator implementation of the sequence map operators allows the skipping and pruning of intermediary results at the earliest possible time if certain additional restrictions are placed on the relations in the query. In particular, we obtain thus $\mathcal{O}(d \cdot q)$ instead of $\mathcal{O}(n \cdot q)$ space bounds for order-compatible queries such as XPath queries containing only child and descendant.

Order-compatible queries enjoy the above improved complexity as they allow us to isolate possible images of a node in a sub-structure of the data *sub-linear* w.r.t. to the data size $n$. However, it is an open question whether this class of queries precisely captures all queries where such an isolation is possible.

### 15.1.3 INTERVAL REPRESENTATION OF ARBITRARY GRAPHS

The sequence map uses interval representations of relations to achieve linear time and space complexity for tree and CIG data when evaluating tree queries. Furthermore, the use of interval representations is useful also for arbitrary graphs (as experimentally verified in Section 14).

However, whether there is a polynomial algorithm for finding the *optimal interval order* or whether the problem is (as we suspect) indeed NP-complete remains an open question.

### 15.1.4 BEYOND INTERVALS: CIQCAG FOR GRAPH QUERIES

The core idea of the CIQCAG algebra is to use the efficient processing of tree queries provided by the sequence map operators and use it to evaluate as much of a given query as possible, before turning to standard relational operators.

In Chapter 13, we use, however, an arbitrary *spanning tree* to separate the tree core from the rest of the query. Better strategies for this separation will be investigated in future work, in particular how optimal tree cores can be approximated efficiently in presence of realistic (global) cost functions. For now, finding an efficient heuristic that yields nearly optimal spanning trees under such cost functions remains an open problem.

The non-tree part of the CIQCAG algebra is currently fairly basic and is a topic of ongoing investigation. In the context of Semantic Web queries, a particularly relevant topic is the *integration of efficient recursion operators*, as surveyed recently in [60].

### 15.1.5 SUPPORTING FULL XPATH, XQUERY, SPARQL, AND XCERPT

Though we show how to translate large fragments of XQuery, SPARQL, XPath, and Xcerpt into CIQLOG and argue that most of the remaining features can be addressed in CIQLOG, full translations require additional work. Considering the promising results from Chapter 14, we believe that, in particular for emerging languages such as SPARQL, CIQCAG-based implementations can considerably outperform existing implementations.

However, to that end both the translation and the current CIQCAG prototype require further work. Of particular interest is also the recent desire to access XML data from SPARQL by means of XPath, a scenario that provides an ideal match for the capabilities of CIQCAG.

### 15.1.6 VERSATILE DATA ACCESS IN THE WEB: A VIRTUAL MACHINE FOR WEB QUERIES

With CIQCAG we have proposed an evaluation for an entire set of Web query languages that is nevertheless capable of evaluating each of these languages with as good a complexity as any previous approach designed specifically for one of these languages.

Thus, we have now the ability to provide a formal basis for the interchange of queries and even (intermediary) result interchange regardless of the surface language either user or data provider prefers. Currently providers dictate the interface language for accessing "their" data: Web

service APIs or "access points" with either proprietary query languages or SPARQL, XPath, or XQuery. Anyone who wants to access that data has to adapt to the provider.

With CIQLOG and CIQCAG, not only is the interface language becoming less important, we can also consider them as first steps towards a common virtual machine for Web queries, similar to the CLI [134] for conventional programming languages.

Naturally, many issues such as security and policies for data access, distribution, etc. are yet to be solved before we can achieve a uniform access layer for Web data.

### 15.1.7 VERSATILE QUERIES FOR BEGINNERS

Finally, though Xcerpt 2.0 has turned out to be a great exemplar for the vision of versatile Web query languages, its use requires, just as in the case of XQuery or even SPARQL, considerable expertise. In an ideal work, a versatile query language should allow the author to formulate his query intent easily and without knowledge about the format the data is stored in. To that end we are currently investigating:

**(1)** *Visual interfaces and languages for versatile Web queries* based on prior work on visXcerpt [23, 25].

**(2)** Drastically simplified query languages in a Semantic Wiki setting (which, to some extent, can be seen as the Web in a really small box) where the author can query the structure of data in the same way, whether it is stored in XML documents or RDF metadata. This work is inspired by and partially founded by the "Knowledge in a Wiki" (KIWI)[1] project.

**(3)** Conceptual query languages are getting increased attention for similar reasons in the Semantic Web context. With CIQLOG and CIQCAG we provide an ideal foundation for realizing such languages where the user can specify the query intent conceptually and the language processor transforms that query intent in specific queries on data in any Web format.

1 http://www.kiwi-project.eu/

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ALGORITHMS

# COMMON ACRONYMS

**AMa$\chi$oS** Abstract Machine for Xcerpt on Semi-structured Data, see Part V

**CIQCAG** Compositional, interval-based Query and Construction Algebra for (Tree and Graph) Queries, see Part IV

**IETF** Internet Engineering Task Force, http://www.ietf.org/

**IRI** Internationalized Resource Identifier, [90]

**ISO** International Organization for Standardization, http://www.iso.org/

**MPEG** Moving Picture Experts Group, http://www.chiariglione.org/mpeg/

**RDF** Application Programming Interface, [150, 142]

**UML** Unified Modeling Language, [165]

**URI** Uniform Resource Identifier, [34]

**W3C** The World Wide Web Consortium, http://www.w3.org/

**XHTML** Extensible HyperText Markup Language, [177]

**XML** Extensible Markup Language, [43]

**XSLT** Extensible Stylesheet Language (XSL) Transformations, [72]

# BIBLIOGRAPHY


[1] Serge Abiteboul and Paris C. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the ACM*, 45(5):798–842, 1998. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/290179.290182. URL http://portal.acm.org/citation.cfm?id=290182.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley Publishing Co., Boston, MA, USA, 1995. ISBN 0-201-53771-0. URL http://db.bell-labs.com/user/hull/FoundDB.html.

[3] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wienerm. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries*, 1(1):68–88, 1997. URL http://www-db.stanford.edu/lore/pubs/lorel96.pdf.

[4] Ben Adida and Mark Birbeck. RDF/A Primer 1.0—Embedding RDF in XHTML. Internal draft, W3C, 2006. URL http://www.w3.org/2001/sw/BestPractices/HTML/2006-01-24-rdfa-primer.

[5] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 253–262, New York, NY, USA, 1989. ACM. ISBN 0-89791-317-5. doi: http://doi.acm.org/10.1145/67544.66950.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974. ISBN 0201000296.

[7] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. Int. Conf. on Data Engineering*, page 141, Washington, DC, USA, 2002. IEEE Computer Society. URL http://www.eecs.umich.edu/~jignesh/publ/xmljoin-ICDE.pdf.

[8] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*

*(SPAA)*, pages 258–264, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7. doi: http://doi.acm.org/10.1145/564870.564914.

[9] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

[10] Sihem Amer-Yahia, Chavdar Botev, Stephen Buxton, Pat Case, Jochen Doerre, Darin McBeath, Michael Rys, and Jayavel Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text. Technical Report Working Draft, W3C, 2005. URL http://www.w3.org/TR/xquery-full-text/.

[11] *plist — Property List Format*. Apple Inc., 2003. URL http://developer.apple.com/documentation/Darwin/Reference/ManPages/man5/plist.5.html.

[12] Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Jendrik Johannes. Modular Web Queries—From Rules to Stores. In *Proc. Int'l. Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2007. URL http://www.pms.ifi.lmu.de/publikationen/. .

[13] Uwe Aßmann, Sacha Berger, François Bry, Tim Furche, Jakob Henriksson, and Paula-Lavinia Pătrânjan. A Generic Module System for Web Rule Languages: Divide and Rule. In *Proc. Int'l. RuleML Symp. on Rule Interchange and Applications*, 2007. URL http://www.pms.ifi.lmu.de/publikationen/. .

[14] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, and Stanley Zdonik. The Object-oriented Database System Manifesto. In François Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-oriented Database System: The Story of O2*, Morgan Kaufmann Series In Data Management Systems, chapter 1, pages 1–20. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992. ISBN 1-55860-169-4. URL http://www.cs.cmu.edu/People/clamen/OODBMS/Manifesto/htManifesto/Manifesto.html.

[15] David Backett. Turtle—Terse RDF Triple Language. Technical report, Institute for Learning and Research Technology, University of Bristol, 2007. URL http://www.dajobe.org/2004/01/turtle/.

[16] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In Jan Małuszyński and Norbert Eisinger, editors, *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, number 3564 in Lecture Notes in Computer Science, pages 35–133. Springer, 2005. .

[17] James Bailey, François Bry, Tim Furche, Benedikt Linse, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Rich Clients need Rich Interfaces: Query Languages for XML and RDF Access on the Web. In *Proc. of German XML-Tage*, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-14.

[18] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. The Elog Web Extraction Language. In *Proc. Int'l. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 548–560, London, UK, 2001. Springer-Verlag. ISBN 3-540-42957-3.

[19] Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format. Proposed recommendation, W3C, 2007. URL http://www.w3.org/TR/rdf-sparql-XMLres/.

[20] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised). Recommendation, W3C, 2004. URL http://www.w3.org/TR/rdf-syntax-grammar/.

[21] Michael Benedikt and Christoph Koch. Interpreting Tree-to-Tree Queries. In *Proc. Int'l. Symp. on Automata, Languages and Programming (ICALP)*, pages 552–564, 2006.

[22] Michael Benedikt and Christoph Koch. XPath Leashed. *ACM Computing Surveys*, 2007.

[23] Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Xcerpt and visXcerpt: Twin Query Languages for the Semantic Web. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-23. .

[24] Sacha Berger, François Bry, Oliver Bolzer, Tim Furche, Sebastian Schaffert, and Christoph Wieser. Querying the Standard and Semantic Web using Xcerpt and visXcerpt. In *Proc. European Semantic Web Conf. (ESWC)*, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-16. .

[25] Sacha Berger, François Bry, and Tim Furche. Xcerpt and visXcerpt: Integrating Web Querying. In *Informal Proc. ACM SIGPLAN*

*Workshop on Programming Language Technologies for XML (PLAN-X)*, page 84, 2006. .

[26] Sacha Berger, François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Beyond XML and RDF: The Versatile Web Query Language Xcerpt. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 1053–1054, 2006. .

[27] Sacha Berger, François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Effective and Efficient Data Access in the Versatile Web Query Language Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 219–224, 2006.

[28] Sacha Berger, François Bry, Tim Furche, and Christoph Wieser. Visual Languages: A Matter of Style. In *Proc. Workshop on Visual Languages and Logic (VLL)*, 2007. URL http://www.pms.ifi.lmu.de/publikationen/.

[29] Sacha Berger, François Bry, Tim Furche, and Andreas J. Häusler. Completing Queries: Rewriting of Incomplete Web Queries under Schema Constraints. In Massimo Marchiori, Jeff Z. Pan, and Christian de Sainte Marie, editors, *Proc. Int'l. Conf. on Web Reasoning and Rule Systems (RR)*, 2007. .

[30] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/77600.77621.

[31] Alexandru Berlea and Helmut Seidl. Binary Queries for Document Trees. *Nordic Journal of Computing*, 11(1):41–71, 2004. URL http://atseidl2.informatik.tu-muenchen.de/~berlea/publications/njc/binaries.pdf.

[32] Tim Berners-Lee. Semantic Web Road Map. Online only, 1998. URL http://www.w3.org/DesignIssues/Semantic.html.

[33] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 2001.

[34] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. Standard RFC 3986, The Internet Society (ISOC) / Internet Engineering Task Force (IETF), 2005.

[35] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2005. URL http://www.w3.org/TR/xquery/.

[36] Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable Logic on Data Trees and XML Reasoning. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 10–19, New York, NY, USA, 2006. ACM. ISBN 1-59593-318-2. doi: http://doi.acm.org/10.1145/1142351.1142354.

[37] Harold Boley. The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. In *Intl. Conf. on Applications of Prolog*, pages 5–22, 2001. URL http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-47086_e.html.

[38] Oliver Bolzer. Towards Data-Integration on the Semantic Web: Querying RDF with Xcerpt. Diplomarbeit/diploma thesis, University of Munich, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#DA_Oliver.Bolzer.

[39] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 479–490, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142527.

[40] Kellogg S. Booth and George S. Lueker. Linear Algorithms to Recognize Interval Graphs and Test for the Consecutive Ones Property. In *Proc. of ACM Symposium on Theory of Computing*, pages 255–265, New York, NY, USA, 1975. ACM Press. doi: http://doi.acm.org/10.1145/800116.803776.

[41] K. A. Bowen and R. A. Kowalski. Amalgamating Language and Metalanguage in Logic Programming. In K. Clark and S. A. Tarnlund, editors, *Logic Programming*, Apic Studies in Data Processing. Academic Press, Inc., 1983.

[42] Daniele Braga, Alessandro Campi, Stefano Ceri, and Enrico Augurusa. XQuery by Example. In *Proc. Int'l. World Wide Web Conf. (WWW)*, 2003. URL http://www2003.org/cdrom/papers/poster/p291/p291-braga.html.

[43] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Third

Edition). Recommendation, W3C, 2004. URL http://www.w3.org/TR/REC-xml/.

[44] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML (2nd Edition). Recommendation, W3C, 2006. URL http://www.w3.org/TR/REC-xml-names/.

[45] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language. Recommendation, W3C, 2004. URL http://www.w3.org/TR/rdf-schema/.

[46] Jeen Broekstra and Arjohn Kampman. An RDF Query and Transformation Language. In *Semantic Web and Peer-to-Peer*, chapter 2, pages 23–39. Springer, 2006.

[47] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16(4):1361–1398, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/183432.183528.

[48] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–321, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-497-5. doi: http://doi.acm.org/10.1145/564691.564727. URL http://www.research.att.com/~divesh/papers/bks2002-twigjoin.pdf.

[49] François Bry, Tim Furche, and Benedikt Linse. Data Model and Query Constructs for Versatile Web Query Languages: State-of-the-Art and Challenges for Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 90–104, 2006.

[50] François Bry, Tim Furche, and Benedikt Linse. AMachoS - Abstract Machine for Xcerpt: Architecture and Principles. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, pages 105–119, 2006.

[51] François Bry, Tim Furche, and Benedikt Linse. Let's Mix It: Versatile Access to Web Data in Xcerpt. In *Proc. of Workshop on Information Integration on the Web (IIWeb)*, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-16.

[52] François Bry, Tim Furche, Benedikt Linse, and Andreas Schroeder. Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In *Proc. ACM Int'l. Workshop on Web Information and*

*Data Management (WIDM)*. ACM Press, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-32. .

[53] François Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. ACM Symp. on Applied Computing (SAC)*. ACM, 2005. URL http://rewerse.net/publications.html#REWERSE-RP-2004-41.

[54] François Bry and Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. Intl. Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, 2002. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2002-11.

[55] François Bry, Tim Furche, and Dan Olteanu. Datenströme. *Informatik Spektrum*, 27(2), 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-2.

[56] François Bry, Tim Furche, Paula-Lavinia Pătrânjan, and Sebastian Schaffert. Data Retrieval and Evolution on the (Semantic) Web: A Deductive Approach. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 3208 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2004. .

[57] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The XML Stream Query Processor SPEX. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 1120–1121, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-1. .

[58] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems*, 1(2), 2005. .

[59] François Bry, Tim Furche, and Sebastian Schaffert. Initial Draft of a Language Syntax (Xcerpt 2.0 Beta). Deliverable I4-D6, Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics), 2006. URL http://rewerse.net/deliverables/m18/i4-d6.pdf.

[60] François Bry, Norbert Eisinger, Thomas Eiter, Tim Furche, Georg Gottlob, Clemens Ley, Benedikt Linse, Reinhard Pichler, and Fang Wei. Foundations of Rule-Based Query Answering. In Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola

Henze, Paula-Lavinia Pătrânjan, and Robert Tolksdorf, editors, *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, number 3564 in Lecture Notes in Computer Science. Springer, 2007. .

[61] François Bry, Tim Furche, Liviu Badea, Christoph Koch, Sebastian Schaffert, and Sacha Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. In Amit Sheth and Miltiadis D. Lytras, editors, *Semantic Web-Based Information Systems: State-of-the-Art Applications*, chapter 8. CyberTech Publishing, 2007. .

[62] François Bry, Tim Furche, Alina Hang, and Benedikt Linse. GRDDLing with Xcerpt: Learn one, get one free! In *Proc. European Semantic Web Conf. (ESWC)*, 2007.

[63] François Bry, Tim Furche, Clemens Ley, and Benedikt Linse. RDFLog: Filling in the Blanks in RDF Querying. Technical Report PMS-FB-2008-01, University of Munich, 2007. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2008-01.

[64] Jan Van Den Bussche, Dirk Van Gucht, Marc Andries, and Marc Gyssens. On the Completeness of Object-creating Database Transformation Languages. *Journal of the ACM*, 44(2):272–319, 1997. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/256303.256311.

[65] Luca Cabibbo. The Expressive Power of Stratified Logic Programs with Value Invention. *Information and Computation*, 147(1):22–56, 1998. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1998.2734.

[66] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named Graphs, Provenance and Trust. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 613–622, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. doi: http://doi.acm.org/10.1145/1060745.1060835.

[67] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[68] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. Working group note, W3C, 2007. URL http://www.w3.org/TR/xquery-use-cases/.

[69] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 77–90, New York, NY, USA, 1977. ACM Press. doi: http://doi.acm.org/10.1145/800105.803397.

[70] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based Algorithms for Pattern Matching on DAGs. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 493–504. VLDB Endowment, 2005. ISBN 1-59593-154-6.

[71] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, and Divesh Srivastava. Index Structures for Matching XML Twigs using Relational Query Processors. *Data & Knowledge Engineering (DKE)*, 60(2):283–302, 2007. ISSN 0169-023X. doi: http://dx.doi.org/10.1016/j.datak.2006.03.003.

[72] James Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999. URL http://www.w3.org/TR/xslt.

[73] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C, 1999. URL http://www.w3.org/TR/xpath/.

[74] Edgar F. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM Transactions on Database Systems*, 4(4): 397–434, 1979. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/320107.320109. URL http://portal.acm.org/citation.cfm?id=320109.

[75] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 451–464, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-560-7. doi: http://doi.acm.org/10.1145/158511.158703.

[76] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proc. ACM Symposium on Discrete Algorithms*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X. URL http://portal.acm.org/citation.cfm?id=545381.545503.

[77] Dan Connolly. Gleaning Resource Descriptions from Dialects of Languages (GRDDL). Recommendation, W3C, 2007. URL http://www.w3.org/TR/grddl/.

[78] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. Int. Conf. on Very Large Databases*, pages 341–350, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4. URL http://www.vldb.org/conf/2001/P341.pdf.

[79] Brouno Courcelle. Graph Rewriting: an Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science*, volume 2, chapter 2, pages 193–242. Elsevier Science Publishers B.V., Cambridge, MA, USA, 1990. ISBN 0-444-88074-7.

[80] Bruno Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[81] John Cowan and Richard Tobin. XML Information Set (2nd Ed.). Recommendation, W3C, 2004. URL http://www.w3.org/TR/xml-infoset/.

[82] Philip T. Cox and Tomasz Pietrzykowski. A Complete, Nonredundant Algorithm for Reversed Skolemization. In *Proc. Int'l. Conf. on Automated Deduction (CADE)*, pages 374–385, London, UK, 1980. Springer-Verlag. ISBN 3-540-10009-1.

[83] Steve DeRose, Eve Maier, and David Orchard. XML Linking Language (XLink) Version 1.0. Recommendation, W3C, 2001. URL http://www.w3.org/TR/xlink/.

[84] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. In *Proc. Int'l. World Wide Web Conf. (WWW)*, 1999. URL http://www.research.att.com/~mff/xmlql/doc/files/final.html.

[85] P. Dietz and D. Sleator. Two Algorithms for Maintaining Order in a List. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, New York, NY, USA, 1987. ACM. ISBN 0-89791-221-7. doi: http://doi.acm.org/10.1145/28395.28434.

[86] Paul F. Dietz. Maintaining Order in a Linked List. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 122–127, New York, NY, USA, 1982. ACM. ISBN 0-89791-070-2. doi: http://doi.acm.org/10.1145/800070.802184.

[87] Andreas Doms, Tim Furche, Albert Burger, and Michael Schroeder. How to Query the GeneOntology. In *Symposium on Knowledge Representation in Bioinformatics (KRBIO)*, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-15. .

[88] Agostino Dovier, Carla Piazza, and Alberto Policriti. An efficient Algorithm for Computing Bisimulation Equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(03)00361-X. URL http://www.dimi.uniud.it/~policrit/Papers/tcsb959.pdf.

[89] Denise Draper, Peter Frankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Recommendation, W3C, 2007. URL http://www.w3.org/TR/xquery-semantics/.

[90] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC (Request for Comments) 3987, IEEE, 2005. URL http://www.faqs.org/rfcs/rfc3987.html.

[91] Ronald Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems*, 2(3):262–278, 1977. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/320557.320571.

[92] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. Recommendation, W3C, 2004. URL http://www.w3.org/TR/xmlschema-0/.

[93] Mary Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing XQuery 1.0 : The Galax Experience. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 2003. URL http://www.vldb.org/conf/2003/papers/S35P07.pdf.

[94] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model. Recommendation, W3C, 2007. URL http://www.w3.org/TR/xpath-datamodel/.

[95] Jörg Flum, Markus Frick, and Martin Grohe. Query Evaluation via Tree-Decompositions. *Journal of the ACM*, 49(6):716–752, 2002. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/602220.602222.

[96] P. Foggia, C. Sansone, and M. Vento. An Improved Algorithm for Matching Large Graphs. In *Proc. Int'l. Workshop on Graph-based Representations*, 2001.

[97] D. R. Fulkerson and O. A. Gross. Incidence Matrices and Interval Graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[98] Tim Furche, François Bry, and Oliver Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 3703 of *Lecture Notes in Computer Science*, pages 72–84. Springer, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-38.

[99] Tim Furche, François Bry, and Oliver Bolzer. XML Perspectives on RDF Querying: Towards integrated Access to Data and Metadata on the Web. In *Proc. GI-Workshop on Grundlagen von Datenbanken*, pages 43–47, 2005. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2005-13.

[100] Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In *Tutorial Lectures Int'l. Summer School 'Reasoning Web'*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006. .

[101] Tim Furche, François Bry, and Sebastian Schaffert. Xcerpt 2.0: Specification of the (Core) Language Syntax. Deliverable I4-D12, Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics), 2007. URL http://rewerse.net/deliverables/m36/i4-d12.pdf.

[102] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[103] Pierre Genevès and Jean-Yves Vion-Dury. XPath Formal Semantics and Beyond: A Coq-Based Approach. In *Proc. Int'l. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 181–198, Salt Lake City, Utah, United States, August 2004. University Of Utah.

[104] R. Gentilini, C. Piazza, and A. Policriti. From Bisimulation to Simulation: Coarsest Partition Problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003. ISSN 0168-7433. doi: http://dx.doi.org/10.1023/A:1027328830731. URL http://www.dimi.uniud.it/~policrit/Papers/JAR_finale.pdf.

[105] Frédérick Giasson and Yves Raimond. Music Ontology Specification. Specification, Zitgist LLC, 2007. URL http://purl.org/ontology/mo/.

[106] Adrian Giurca and Dorel Savulea. An Algebra of Logic Programs with Applications in Distributed Environments. In *Annales of*

*Craiova University*, volume XXVIII of *Mathematics and Computer Science Series*, pages 147–159, 2001.

[107] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7.

[108] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 21–32, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-062-7. doi: http://doi.acm.org/10.1145/303976.303979.

[109] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 21–32, New York, NY, USA, 1999. ACM. ISBN 1-58113-062-7. doi: http://doi.acm.org/10.1145/303976.303979.

[110] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The Complexity of Acyclic Conjunctive Queries. *Journal of the ACM*, 48(3): 431–498, 2001. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/382780.382783. URL http://portal.acm.org/citation.cfm?id=382783.

[111] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[112] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 2005. URL http://www.infosys.uni-sb.de/~koch/download/tods1.pdf.

[113] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *Journal of the ACM*, 2005. URL http://www-db.cs.uni-sb.de/~koch/download/jacm2.pdf.

[114] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/152610.152611.

[115] Martin Grohe and Nicole Schweikardt. Comparing the Succinctness of Monadic Query Languages over Finite Trees. In *Proc. Workshop on Computer Science Logic (CSL)*, 2003.

[116] Thorsten Grust. Accelerating XPath Location Steps. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, 2002. URL http://www.in.tu-clausthal.de/~grust/files/xpath-accel.pdf.

[117] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue - XQuery: Fluent. In *Proc. Twente Data Management Workshop on XML Databases and Information Retrieval*, 2004. URL http://www.in.tu-clausthal.de/~grust/files/algebra-mapping.pdf.

[118] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach A Relational DBMS to Watch its (Axis) Steps. In *Proc. Int. Conf. on Very Large Databases*, 2003. URL http://www.inf.uni-konstanz.de/~teubner/publications/watch-axis-steps.pdf.

[119] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/974750.974754.

[120] Michel Habib, Ross McConnell, Christophe Paul, and Laurent Viennot. Lex-BFS and Partition Refinement, with Applications to Transitive Orientation, Interval Graph Recognition and Consecutive Ones Testing. *Theoretical Computer Science*, 234(1-2):59–84, 2000. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(97)00241-7.

[121] Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/0213024.

[122] Patrick Hayes and Brian McBride. RDF Semantics. Recommendation, W3C, 2004.

[123] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with Aggregate Operators. *Journal of the ACM*, 48(4):880–907, 2001. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/502090.502100.

[124] Jakob Henriksson, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Reuseware—Adding Modularity to Your Language of Choice. *Journal of Object Technology*, 6(9 (Special Issue. TOOLS EUROPE 2007)):127–146, 2007.

[125] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proc. Symp. on Foundations of Computer Science (FOCS)*, page 453, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7183-1.

[126] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. ACM Symposium on Theory of Computing*, pages 172–184, New York, NY, USA, 1974. ACM Press. doi: http://doi.acm.org/10.1145/800119.803896.

[127] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grossof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Member submission, W3C, 2004. URL http://www.w3.org/Submission/SWRL/.

[128] Sun-Yuan Hsieh. The Interval-Merging Problem. *Information Systems*, 177(2):519–524, 2007.

[129] Wen-Lian Hsu. PC-Trees vs. PQ-Trees. In *Proc. Int'l. Conf. on Computing and Combinatorics*, volume 2108 of *LNCS*, 2001.

[130] Wen-Lian Hsu. A Simple Test for the Consecutive Ones Property. *Journal of Algorithms*, 43(1):1–16, 2002. ISSN 0196-6774. doi: http://dx.doi.org/10.1006/jagm.2001.1205.

[131] Wen-Lian Hsu and Ross M. McConnell. PC Trees and Circular-ones Arrangements. *Theoretical Computer Science*, 296(1):99–116, 2003. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(02)00435-8.

[132] Richard Hull and Masatoshi Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 455–468, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. ISBN 0-55860-149-X.

[133] *ISO/IEC 13250 Topic Maps*. International Organization for Standardization (ISO), 2nd edition, 2002. URL http://www1.y12.doe.gov/capabilities/sgml/sc34/document/0322.htm.

[134] ISO/IEC. 23271, Common Language Infrastructure (CLI). International Standard 23271, ISO/IEC, 2003.

[135] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Proc. Int. Workshop on Database Programming Languages*, 2001. URL http://www.cs.ubc.ca/~laks/tax-dbpl01-cr.pdf.

[136] H. Jiang, H. Lu, W. Wang, and B. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Join. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 253–264, 2003. URL citeseer.ist.psu.edu/jiang03xrtree.html.

[137] Isambo Karali, Evangelos Pelecanos, and Constantin Halatsis. A Versatile Module system for Prolog Mapped to Flat Prolog. In *Proc. ACM Symp. on Applied Computing (SAC)*, pages 578–585, New York, NY, USA, 1993. ACM Press. ISBN 0-89791-567-4. doi: http://doi.acm.org/10.1145/162754.168687.

[138] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a Declarative Query Language for RDF. In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 592–603, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-449-5. doi: http://doi.acm.org/10.1145/511446.511524.

[139] Michael Kay. XSL Transformations, Version 2.0. Recommendation, W3C, 2007.

[140] Setrag N. Khoshafian and George P. Copeland. Object Identity. In *Proc. Intl. Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 406–416, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-204-7. doi: http://doi.acm.org/10.1145/28697.28739. URL http://portal.acm.org/citation.cfm?id=28739.

[141] Michael Kifer, Jos de Bruijn, Harold Boley, and Dieter Fensel. A Realistic Architecture for the Semantic Web. In *Proc. Int'l. Conf. on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, volume 3791 of *LNCS*, pages 17–29. Springer, 2005.

[142] Graham Klyne, Jeremy J. Carroll, and Brian McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax. Recommendation, W3C, 2004.

[143] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser/Springer-Verlag, 1993.

[144] Christoph Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *tods*, 31(4), 2006. URL http://www.infosys.uni-sb.de/~koch/download/0503062.pdf.

[145] Gabriel M. Kuper and Moshe Y. Vardi. The Logical Data Model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/155271.155274. URL http://portal.acm.org/citation.cfm?id=155274.

[146] Dirk Leinders, Maarten Marx, Jerzy Tyszkiewicz, and Jan Van den Bussche. The Semijoin Algebra and the Guarded Fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005.

[147] Leonid Libkin and Limsoon Wong. Query Languages for Bags and Aggregate Functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997. ISSN 0022-0000. doi: http://dx.doi.org/10.1006/jcss.1997.1523.

[148] Benedikt Linse. Automatic Translation between XQuery and Xcerpt. Diplomarbeit/diploma thesis, Institute for Informatics, University of Munich, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#DA_Benedikt.Linse.

[149] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Recommendation, W3C, 2007. URL http://www.w3.org/TR/xpath-functions/.

[150] Frank Manola, Eric Miller, and Brian McBride. RDF Primer. Recommendation, W3C, 2004.

[151] Jonathan Marsh. XML Base. Recommendation, W3C, 2001. URL http://www.w3.org/TR/xmlbase/.

[152] José M. Martínez. MPEG-7 Overview. Technical Report ISO/IEC JTC1/SC29/WG11N6828, INTERNATIONAL ORGANISATION FOR STANDARDISATION (ISO), 2004. URL http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm.

[153] Maarten Marx. First Order Paths in Ordered Trees. In *Proc. Int'l. Conf. on Database Theory (ICDT)*, pages 114–128, 2005. URL http://staff.science.uva.nl/~marx/pub/recent/icdt05.pdf.

[154] Maarten Marx. Conditional XPath, the First Order Complete XPath Dialect. In *Proc. ACM Symposium on Principles of Database Systems*, pages 13–22. ACM, 6 2004. URL http://turing.wins.uva.nl/~marx/pub/recent/pods04.pdf.

[155] Norman May, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. XQuery Processing in Natix with an Emphasis on Join Ordering. In *Proc. of Int. Workshop on XQuery Implementation, Experience and Perspectives*,

2004. URL http://pi3.informatik.uni-mannheim.de/old/publications/ximep2004-joinorder.ps.

[156] Wolfgang May. XPath-Logic and XPathLog: A Logic-Programming Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 3(4):499–526, 2004.

[157] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language—Overview. Recommendation, W3C, 2004. URL http://www.w3.org/TR/owl-features/.

[158] B. D. McKay. Practical Graph Isomorphism. In *Proc. Conf. on Numerical Mathematics and Computing*, 1980.

[159] Jim Melton. *Advanced SQL: 1999—Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers Inc., 2002.

[160] Holger Meuss and Klaus U. Schulz. Complete Answer Aggregates for Treelike Databases: A Novel Approach to Combine Querying and Navigation. *ACM Transactions on Information Systems*, 19(2):161–215, 2001. ISSN 1046-8188. doi: http://doi.acm.org/10.1145/382979.383042. URL http://www.cis.uni-muenchen.de/people/Meuss/Pub/TOIS01.pdf.

[161] Holger Meuss, Klaus U. Schulz, and François Bry. Towards Aggregated Answers for Semistructured Data. In *Proc. Intl. Conf. on Database Theory*, pages 346–360. Springer-Verlag, 2001. ISBN 3-540-41456-8. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2000-15.

[162] Dale Miller. A Theory of Modules for Logic Programming. In *Proc. IEEE Symp. on Logic Programming*, pages 106–114, 1986.

[163] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, June 2002.

[164] Robin Milner. An Algebraic Definition of Simulation Between Programs. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[165] Object Management Group. UML 2.0 Superstructure Specification. Specification, Object Management Group, 2005. URL http://www.omg.org/technology/documents/formal/uml.htm.

[166] Frank Olken and John McCarthy. Requirements and Desiderata for an XML Query Language. In *Proc. W3C QL'98 – Query Languages 1998*, December 1998.

[167] Dan Olteanu. Forward Node-selecting Queries over Trees. *ACM Transactions on Database Systems*, 32(1):3, 2007. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/1206049.1206052.

[168] Dan Olteanu. SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 2007.

[169] Dan Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/doctoral thesis, University of Munich, 2005.

[170] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *Lecture Notes in Computer Science*. Springer, 2002. .

[171] Dan Olteanu, Tim Furche, and François Bry. Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In *Proc. British National Conf. on Databases (BNCOD)*, pages 31–44, 2003. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2003-15. .

[172] Dan Olteanu, Tim Furche, and François Bry. An Efficient Single-Pass Query Evaluator for XML Data Streams. In *Data Streams Track,Proc. ACM Symp. on Applied Computing (SAC)*, pages 627–631, 2004. .

[173] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML Node Labels. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 903–908. ACM Press, 2004. ISBN 1-58113-859-8. doi: http://doi.acm.org/10.1145/1007568.1007686. URL http://www.cs.umb.edu/~poneil/ordpath.pdf.

[174] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987. ISSN 0097-5397. doi: http://dx.doi.org/10.1137/0216062.

[175] Jack Park and Sam Hunting, editors. *XML Topic Maps: Creating and Using Topic Maps for the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201749602.

[176] Peter Patel-Schneider and Jerome Simeon. The Yin/Yang Web: XML Syntax and RDF Semantics. In *Proc. Intl. World Wide Web Conference*, May 2002. URL http://www2002.org/CDROM/refereed/231/.

[177] Steven Pemberton and the W3C HTML Working Group. XHTML 1.0: The Extensible HyperText Markup Language. Recommendation, W3C, 2000.

[178] Steve Pepper, Fabio Vitali, Lars Marius Garshol, Nicola Gessa, and Valentina Presutti. A Survey of RDF/Topic Maps Interoperability Proposals. Working group note, W3C, 2006. URL http://www.w3.org/TR/rdftm-survey/.

[179] Jorge Perez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2006.

[180] Alexander Pohl. RDF Querying in Xcerpt: Language Constructs and Implementation. Diplomarbeit/diploma thesis, University of Munich, 2008.

[181] Axel Polleres. From SPARQL to Rules (and Back). In *Proc. Int'l. World Wide Web Conf. (WWW)*, pages 787–796, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: http://doi.acm.org/10.1145/1242572.1242679.

[182] Eric Prud'hommeaux. RDF Data Access Working Group Charter (W3C). Online only, 2003. URL http://www.w3.org/2003/12/swa/dawg-charter.

[183] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Proposed recommendation, W3C, 2007. URL http://www.w3.org/TR/rdf-sparql-query/.

[184] Jonathan Robie. The Syntactic Web. In *Proc. XML Conference and Exhibition*, 2001. URL http://www.idealliance.org/papers/xml2001/papers/html/03-01-04.html.

[185] Loïc Royer, Benedikt Linse, Thomas Wächter, Tim Furche, François Bry, and Michael Schroeder. Querying the Semantic Web: A Case Study. In *Revolutionizing Knowledge Discovery in the Life Sciences*. Springer, 2006.

[186] D. T. Sannella and L. A. Wallen. A Calculus for the Construction of Modular Prolog Programs. *Journal of Logic Programming*, 12(1-2):147–177, 1992. ISSN 0743-1066. doi: http://dx.doi.org/10.1016/0743-1066(92)90042-2.

[187] Sebastian Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/doctoral thesis, University of Munich, 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-DISS-2004-1.

[188] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages (Int'l. Conf. on Markup Theory & Practice)*, 2004. URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2004-7.

[189] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proc. Extending Database Technology*, 2004. URL http://wwwcs.uni-paderborn.de/cs/ag-boettcher/lehre/SS04/seminar/download/edbt04.HOPI.An.Efficient.Connection.Index.for.Complex.XML.Document.Collections.pdf.

[190] Mirit Shalem and Ziv Bar-Yossef. The Space Complexity of Processing XML Twig Queries over Indexed Documents. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2008.

[191] Yeh-Heng Shen. IDLOG: Extending the Expressive Power of Deductive Database Languages. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 54–63, New York, NY, USA, 1990. ACM. ISBN 0-89791-365-5. doi: http://doi.acm.org/10.1145/93597.93621.

[192] David W. Shipman. The Functional Data Model and the Data Languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/319540.319561. URL http://portal.acm.org/citation.cfm?id=319561.

[193] Michael Sintek and Stefan Decker. TRIPLE—A Query, Inference, and Transformation Language for the Semantic Web. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2002.

[194] Damian Steer. TreeHugger 1.0 Introduction. Online only, 2003. URL http://www.semanticplanet.com/2003/08/rdft/spec.

[195] Silke Trißl and Ulf Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *Proc. ACM Symp. on Management of Data*

*(SIGMOD)*, pages 845–856, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: http://doi.acm.org/10.1145/1247480.1247573.

[196] M. Vardi. The Complexity of Relational Query Languages. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 137–146, San Francisco, 1982.

[197] Norman Walsh. RDF Twig: accessing RDF graphs in XSLT. In *Proc. Extreme Markup Languages*, 2003. URL http://www.mulberrytech.com/Extreme/Proceedings/xslfo-pdf/2003/Walsh01/EML2003Walsh01.pdf.

[198] Norman Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O?Reilly, 1999. URL http://www.oreilly.com/catalog/docbook/.

[199] Haixun Wang, Hao He2, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, page 75, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: http://dx.doi.org/10.1109/ICDE.2006.53.

[200] Felix Weigel, Klaus U. Schulz, and Holger Meuss. The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations. In *Proc. Int'l. XML Database Symposium (XSym)*, volume 3671 of *LNCS*, pages 49–67. Springer-Verlag, 2005.

[201] Artur Wilk and Wlodzimierz Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Proc. Int'l. Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, volume 4187 of *LNCS*, pages 262–275. Springer, 2006. URL http://www.pms.ifi.lmu.de/publikationen/#REWERSE-RP-2006-043.

[202] Martin Wirsing. Structured Algebraic Specifications: A Kernel Language. *Theoretical Computer Science*, 42(2):123–244, 1986. ISSN 0304-3975.

[203] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Proc. Int. Conf. on Very Large Data Bases*, pages 82–94, 1981.

[204] Moshe M. Zloof. Query By Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.

[205] Moshé M. Zloof. Query By Example. In *AFIPS National Computer Conference*, 1975.

## ABOUT THE AUTHOR

I have a passion for thinking about, designing, and realizing next generation Web systems. This passion has driven and still drives most of my research interests and activities. Within this general topic, my interest lies mostly in efficient access to Web data through declarative query languages or similar facilities. Rich declarative interfaces make it possible to access and reuse Web data in novel ways (cf. mash-ups). I have worked on practical aspects of efficient query evaluation against unbounded streams, an enabling technology for the (near real-time) detection of alarm conditions in, e.g., sensor streams. More recently, my focus has been on the development of versatile Web query languages, in particular on the refinement as well as the study of theoretical properties and efficient evaluation of Xcerpt, a declarative, logic-based Web query language.

### CURRICULUM VITAE

| | |
|---|---|
| 1976 | born in Tübingen, Baden-Württemberg, Germany |
| 1996 | High-school graduation (Abitur) with distinction in computer science |
| 1998-2003 | Diploma student in computer science and computational linguistics at the Ludwig-Maximilians Universität München, supported by a scholarship of the German National Academic Foundation |
| 2003 | Diploma graduation with distinction with thesis on "Optimizing Multiple Queries against XML Streams" |
| 2004-2008 | Research assistant at chair of Prof. François Bry, Ludwig-Maximilians Universität München, as part of the EU Network of Excellence REWERSE (assistant coordinator working group "Reasoning-aware Querying") |
| 2008-now | Research and teaching assistant at chair of Prof. François Bry |

## LEBENSLAUF

| | |
|---|---|
| | Tim Jakob Furche |
| | geboren am 3. Dezember 1976 in Tübingen, Baden-Württemberg, Deutschland |
| 1996 | Abitur mit Auszeichnung in Informatik |
| 1997 | Datenbankentwickler an der Diakonissenanstalt Stuttgart, Deutschland |
| 1998-2003 | Diplomstudium in Informatik mit Nebenfach Computerlinguistik an der Ludwig-Maximilians Universität München mit einem Stipendium der Studienstiftung des Deutschen Volkes |
| 2003 | Diplomprüfung mit Auszeichnung, Thema der Diplomarbeit "Optimizing Multiple Queries against XML Streams" |
| 2004-2008 | Wissenschaftlicher Mitarbeiter am Lehrstuhl Prof. François Bry, Ludwig-Maximilians Universität München, im Rahmen des EU Network of Excellence REWERSE (Assistant coordinator der Arbeitsgruppe "Reasoning-aware Querying") |
| 2008-now | Wissenschaftlicher Mitarbeiter am Lehrstuhl Prof. François Bry |