University of Twente.

Master Thesis

Database Group

Efficient Query Evaluation on Probabilistic XML Data

Derived from a glue process with skeleton & flesh

Paul Stapersma
5th December 2012

Committee:
Dr. M. Van Keulen (UT/DB)
Dr. M. M. Fokkinga (UT/DB)
Ing. J. Flokstra (UT/DB)

"All the ideas in the universe can be described by words. Therefore, if you simply take all the words and rearrange them randomly enough times, you're bound to hit upon at least a few great ideas eventually."

– Jarod Kintz

Abstract

In many application scenarios, reliability and accuracy of data are of great importance. Data is often uncertain or inconsistent because the exact state of represented real world objects is unknown. A number of uncertain data models have emerged to cope with imperfect data in order to guarantee a level of reliability and accuracy. These models include probabilistic XML (P-XML) –an uncertain semi-structured data model– and U-Rel –an uncertain table-structured data model. U-Rel is used by MayBMS, an uncertain relational database management system (URDBMS) that provides scalable query evaluation. In contrast to U-Rel, there does not exist an efficient query evaluation mechanism for P-XML.

In this thesis, we approach this problem by instructing MayBMS to cope with P-XML in order to evaluate XPath queries on P-XML data as SQL queries on uncertain relational data. This approach entails two aspects: (1) a data mapping from P-XML to U-Rel that ensures that the same information is represented by database instances of both data structures, and (2) a query mapping from XPath to SQL that ensures that the same question is specified in both query languages.

We present a specification of a P-XML to U-Rel data mapping and a corresponding XPath to SQL mapping. Additionally, we present two designs of this specification. The first design constructs a data mapping in such way that the corresponding query mapping is a traditional XPath to SQL mapping. The second design differs from the first in the sense that a component of the data mapping is evaluated as part of the query evaluation process. This offers the advantage that the data mapping is more efficient. Additionally, the second design allows for a number of optimizations that affect the performance of the query evaluation process. However, this process is burdened with the extra task of evaluating the data mapping component.

An extensive experimental evaluation on synthetically generated data sets and real-world data sets shows that our implementation of the second design is more efficient in most scenarios. Not only is the P-XML data mapping executed more efficient, the query evaluation performance is also improved in most scenarios.

Preface

As a scholar, I had a wide interest for many specialties such as finance, physics and mathematics. Consequently, I had no idea what study would intrigue me the most. I participated in a promotion project in which I was accompanied by a senior student who showed me his daily life at the campus in Enschede. This opportunity resulted in me becoming a Computer Science (CSC) student at the University of Twente.

In my first year as student, I came in contact with various interesting fields of computer science such as telematics, security and databases. As a result, I started in the same year with a second Bachelor's program in Telematics. Additionally, I participated in extracurricular activities and became member of the CSC promotion team. This time, it was my turn to show scholars the student life.

At the end of my bachelor, I was asked to introduce a reporter to several researchers in the field of CSC. During this activity, I came in contact with Maurice van Keulen, my first supervisor of this graduation project. He sketched the reporter his field of research by which he indirectly introduced me to the field of uncertain databases. At that time, I had to select a topic for my final Bachelor project. I asked Maurice if I could participate in one of his research projects as part of my Bachelor project. This was the start of a wonderful collaboration.

I continued my study with a master in security. This turned out to be a bad match. After a switch from security to databases, I had to pleasure to work with Maurice once again on two projects that build on my initial Bachelor project. The rough diamonds we found during these projects were the input for this graduation project.

During my graduation project, many people asked me what my research is about. Most of the times, I try to explain the concept of an uncertain database management system and sometimes I add an application scenario to this explanation. One day, I was walking with my dad in the park. He told me that I had to find an application scenario that had to be appealing to people. The next two weeks, I found myself building a solver for nonogram puzzles with solely URDBMS technology. One solution to such puzzle is found in Figure 1. Unfortunately, I was unable to put my thoughts of this new idea on paper. However, this finding has convinced me that URDBMS technology has a promising future.

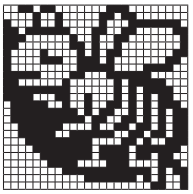


Figure 1: Illustration of a nonogram

Acknowledgements

I would like to thank a few people for their support during the course of my graduation project in which this thesis has been written. First of all, I would like to thank my supervisors: Maurice, Maarten en Jan. Maurice, I really appreciate the freedom you gave me to mastermind my own thoughts and help me conquer most of the challenges in this research project and earlier projects. I will miss our long discussions and brainstorm sessions about how to take our projects to the next level. Maarten, you amazed me with your skills to put a complex idea on paper in just a few lines. In the time we spent, you taught me the basics of how to formalize my own ideas. The 26th letter of the alphabet will always help me remind me of this. Jan, thank you for all the support on realizing a full grown P-XML DBMS prototype & benchmark. Also your crash course in C helped me master MayBMS.

I would also like to thank my fellow year students: Lesley Wevers, Harold Bruintjes, Ronald Burgman, Gerjan Stokkink, Björn Postema, Ferry Olthuis and Daan van Beek. They have provided me with a pleasant environment at the fifth flour. I like to acknowledge Harold in particular for his contributions to the image processing in this thesis, the high-fives and the many coffee breaks.

I would like to thank Matthias Bosch for helping me getting my thoughts on paper. I experienced that the gap between knowing something and explaining something can be huge. Matthias helped me bridge this gap.

Finally, I would like to thank my friends and family for supporting me. Especially my brother who helped me visualize nonogram solving.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Introduction to uncertainty management . . . 1
  - 1.1.2 Application scenarios of uncertainty management . . . 2
  - 1.1.3 Uncertainty management for XML . . . 3
- 1.2 Research questions . . . 3
- 1.3 Global approach to show correctness . . . 4
- 1.4 High level design . . . 7
- 1.5 Scope . . . 8
  - 1.5.1 Suitable URDBMS for XPath processing . . . 8
  - 1.5.2 Probabilistic XML model . . . 8
  - 1.5.3 XPath support . . . 8
- 1.6 Contributions . . . 9
- 1.7 Outline . . . 9

I Prologue **11**

2 Preliminaries **13**
- 2.1 An abstract view on database mappings . . . 13
- 2.2 Introduction to XML . . . 13
  - 2.2.1 Extensible Markup Language . . . 13
  - 2.2.2 XPath expressions . . . 14
- 2.3 XML into RDBMS mappings . . . 15
  - 2.3.1 Shared Inlining, a schema-based mapping of XML to relational databases . . . 16
  - 2.3.2 XPath Accelerator, a schema-less mapping of XML to relational databases 18
- 2.4 Introduction to uncertain databases . . . 21
  - 2.4.1 Interpretation of an uncertain database in terms of possible worlds . . . 21
  - 2.4.2 Granularity of uncertainty . . . 21
- 2.5 Probabilistic XML, an uncertain semi-structured data model . . . 21
  - 2.5.1 P-XML model of Van Keulen et al. . . . 22
  - 2.5.2 Query evaluation on P-XML data . . . 23
- 2.6 Views, unmaterialized and materialized . . . 23

3 From P-XML to U-Rel, the Detour **25**
- 3.1 From P-XML to C-XML . . . 26
- 3.2 From C-XML to Unode . . . 27
- 3.3 From Unode to Urow . . . 29
- 3.4 From Urow to URel . . . 29
- 3.5 Summary . . . 32

II Specification **35**

4 Abstraction of an Uncertain Data Model **37**
- 4.1 Analogy with pattern matching . . . 37
- 4.2 An abstract formalism of a data model . . . 38
- 4.3 The U-Relational model adheres to the possible worlds semantics . . . 43
- 4.4 Analogy with pattern matching continued . . . 44
- 4.5 An abstract formalism of an uncertain data model . . . 45
- 4.6 Query evaluation on uncertain data . . . 47
- 4.7 $\mathcal{U}$ in the big picture . . . 50

4.8 Summary . . . 51

5 Probabilistic XML expressed in an Abstract Formalism 53
5.1 Probabilistic XML data structure . . . 53
5.2 P-XML adheres to the possible worlds semantics . . . 55
5.3 Viability . . . 56
5.4 C-XML data structure . . . 57
5.5 P-XML to C-XML mapping . . . 58
5.5.1 Data mapping . . . 58
5.5.2 Query mapping . . . 59
5.6 C-XML into U mapping . . . 60
5.6.1 Data mapping . . . 60
5.6.2 Query mapping . . . 61
5.7 Summary . . . 62

6 U-Rel expressed in an Abstract Formalism 63
6.1 U-Relational model . . . 63
6.1.1 U-Relations adhere to the possible world semantics . . . 64
6.2 U-Rel expressed in $\mathcal{U}$ . . . 64
6.2.1 Data mapping . . . 64
6.2.2 Query mapping . . . 65

7 Mapping of Nodes to Rows 67
7.1 Data mapping . . . 67
7.2 Query mapping . . . 68

III Design 69

8 P-XML into URDBMS Data Mapping 71
8.1 Construct a c-document from flesh & skeleton . . . 71
8.2 First design of P-XML into URDBMS mapping from flesh & skeleton . . . 72
8.3 Mapping of the skeleton into a URDBMS . . . 73
8.4 Mapping of the flesh into a URDBMS . . . 73
8.4.1 Flesh mapping requirements . . . 73
8.4.2 ASI[XA], a new XML into RDBMS mapping designed as flesh mapping . . . 74
8.5 Summary . . . 76

9 Introduction to Relational Gluing 77
9.1 Phases . . . 77
9.2 Dependency handles . . . 77
9.3 G-Join . . . 78
9.4 Don't care choice . . . 79
9.5 Depth . . . 80
9.6 Summary . . . 81

10 Glue Processes 83
10.1 Flavours of glue method application . . . 83
10.1.1 Batch based application . . . 83
10.1.2 Partition based application . . . 84
10.1.3 Precomputed chaining . . . 84
10.2 Categories . . . 85
10.2.1 Flesh driven glue methods . . . 86
10.2.2 Skeleton driven glue methods . . . 90
10.2.3 Inheritance driven glue methods . . . 93

10.3 Glue administering . . . 94
10.3.1 Document oriented gluing, Administering a glue process to the data mapping process . . . 95
10.3.2 Query result oriented gluing, Administering a glue process to the query mapping process . . . 96
10.3.3 Qualitative comparison between document oriented and query result oriented gluing . . . 97
10.4 Tree-aware uncertainty management . . . 97
10.5 Summary . . . 100

IV Validation 101

11 Overview of MayBMS & Optimizations 103
11.1 Repair-key statement . . . 103
11.1.1 From inconsistency to uncertainty . . . 103
11.1.2 Implementation details . . . 104
11.1.3 Small benchmark on repair-key statement . . . 104
11.1.4 Multi-union approach . . . 105
11.1.5 Benchmark on multi-union approach applied to repair-key . . . 108
11.2 XPath Accelerator vs. Abstract Shared Inlining . . . 110
11.3 Glue methods . . . 111

12 Experiments 113
12.1 Goal . . . 113
12.2 Experimental setup . . . 113
12.2.1 Test sets . . . 113
12.2.2 Query workload . . . 115
12.2.3 Database optimizations . . . 116
12.2.4 Test platform . . . 116
12.2.5 Obtaining benchmark results . . . 116
12.3 Experimental results . . . 118
12.3.1 Experimental results for P-XML into URDBMS data mappings . . . 118
12.3.2 Experimental results for XPath evaluation by a URDBMS . . . 122
12.3.3 Conclusions . . . 128

13 Related Work 131

14 Conclusions & Future Work 133
14.1 Summary . . . 133
14.2 Evaluation of research questions . . . 134
14.3 Research goal achievement . . . 135
14.4 Future work . . . 135

Bibliography 136

A Proofs 141
A.1 Closest node . . . 141

B SQL queries to perform glue process 143
B.1 Glue by Possibility Parent Reference . . . 143
B.2 Glue by Sandwich . . . 143
B.3 Glue by Closest Descendant . . . 144
B.4 Glue by Depth . . . 144

C Benchmark Details **145**
C.1 Query Workload 145
C.2 Indices for query evaluation 147
C.3 DTD of IMDb data set 152
C.4 Normalized speedups for XPath query evaluation by a URDBMS 152
C.4.1 Experimental results for query evaluation on increasing data set size 152
C.4.2 Experimental results for query evaluation on increasing amounts of uncertainty 154
C.4.3 Experimental results for query evaluation on a real world uncertain test set 154

D Retrospect on expressiveness **157**

Acronyms **159**

Glossary **161**

Chapter 1

Introduction

1.1 Motivation

In many application scenarios, reliability and accuracy of data are of great importance. Data is often uncertain or inconsistent because the exact state of represented real world objects is unknown. Therefore, data imperfections have to be managed by information systems in order to guarantee a level of data quality. One way to accomplish this is with uncertainty management. Uncertainty management allows an information system to cope with data that is imperfect. We provide an introduction to uncertainty management in Section 1.1.1.

In addition, uncertainty management lends itself for other applications like using user feedback in data management systems in order to improve data quality or trustworthiness of information systems. We elaborate on the diversity of applications for uncertainty management in Section 1.1.2.

In many application scenarios of uncertainty management, information is described in a semi-structured data model. As a consequence, research introduced several probabilistic XML (P-XML) data models that allow for uncertain semi-structured data storage. Section 1.1.3 provides a more detailed motivation for uncertain semi-structured data models. We claim that the state of art does not provide an efficient query evaluation mechanism for P-XML that is scaled up to practice. This is the main motivation for our approach to build an efficient query evaluation mechanism for P-XML data.

1.1.1 Introduction to uncertainty management

In many application domains, data is generally assumed to be complete, correct and conform to reality. These idealistic assumptions are reflected by the expectations of users, who presume their systems to know everything they want to know, and developers who design their systems to be based on perfect data. It is unrealistic to live up to these expectations since a lot of data generally contains many types of imperfections.

A survey on uncertainty management [39] classified several classes of *data imperfection*. We borrowed their example to illustrate these classes which are found in Table 1.1. Various types of data imperfection may coexist, such as in: *John is probably not very tall*. The author noted that the names assigned to the different classes are used in many existing taxonomies of imperfection, however, slightly different classifications are used in other works.

Class	Example: John's tallness
No imperfection	183cm.
Absence/Missing values	Not known.
Non-specificity	Between 180 and 190cm.
	183 or 184 or 185cm.
Vagueness	Not very tall.
Uncertainty	Perhaps, 183cm.
Inconsistency	183 and 184 and 185cm.
Error	170cm.

Table 1.1: The main recognized classes of data imperfection

Reasons why data is inexact or not reliable could be one of the following: (1) some data is inexact due to the nature of its origin, (2) data derived from inexact data is also inexact, (3) decisions cannot always be made with only the data at hand, by which a system is forced to make

an educated guess with all the consequences that will entail, (4) statistical operations give results with some probability, (5) an approximate answer close to the exact answer can be computed quickly while the exact answer can be computed in the background or not at all in case the approximate answer is sufficient [52].

By its very nature, data imperfections affect the reliability and accuracy of a data source. Hence, they have to be managed in a sensible way. As argued by Halevy [25], standard data management tasks should include a notion of accuracy and reliability in order to provide a level of data quality. We refer to this kind of management as *uncertainty management*.

The terminology uncertainty management seems misplaced, since uncertainty management implies to manage only uncertainty imperfections, while it should give a notion of the reliability and accuracy of data. However, inconsistency can be interpreted as being uncertain about which of the conflicting values is correct [25, 49]. A similar interpretation can be applied to the discrete case of the non-specificity imperfection class in case only one value is known to be correct. Hence, many classes of data imperfection can be managed. The term 'data quality management' would seem more suitable, since more classes of data imperfection are managed with uncertainty management than solely the uncertain class.

If we return to the example in Table 1.1, we can treat the inconsistency in *John's tallness is '183 and 184 and 185 cm.*' as *John's tallness is* '*Perhaps, 183cm.*' or '*Perhaps 184cm.*' or '*Perhaps 185cm*'. We can apply a similar treatment to the example of the *non-specificity* class with the knowledge that John only has one single tallness to obtain the same result.

1.1.2 Application scenarios of uncertainty management

Reliability and accuracy of data are of great importance in many application domains. Inexact data can be enriched with self-describing information about their reliability or accuracy, called uncertain data. The use of uncertain data can be exploited in several application domains. Widom [52] mentions the following candidates: scientific data management, sensor data management, data deduplication, profile assembly, privacy preservation, approximate query processing, hypothetical reasoning and online query processing.

According to Halevy [25], uncertainty management is one of the challenges that arise in enterprise and government data management as a result of system architectures characterized by loosely connected heterogeneous data sources.

Lynch [38] argues that uncertainty management should also be applied to information retrieval systems which deal with databases that are only assumed to be trustworthy and accurate, and are treated as such. Uncertainty management should indicate to what extent these assumptions are correct.

By its very nature, uncertain data allows systems to manage multiple states. Such a property can be very useful in application scenarios where hard decisions have to be made with little information at hand, because the decision making process can be postponed until sufficient information is available. In the meantime, multiple states are managed, one for each possible outcome of the decision. Examples of application scenarios that use uncertain data for the postponing of hard decisions are duplicate detection: the detection of duplicate tuples corresponding to the same real-world entity [4, 42, 49], named entity disambiguation [24], information extraction [49, 31], data cleaning [13], data coupling/fusion [49], data integration [50], natural language processing: interpreting a natural language by building a syntax tree out of sentences [40, 15].

Most promising seems the integration of user feedback functionality with data management systems that support uncertainty management. The ability of users to interact with a data management system can greatly improve data quality as demonstrated by Kuperus [36] and Van Keulen et al. [50]. This field of research is identified by Halevy [25] as a key tenet that allows data management systems to evolve by learning from human attention. Halevy referred to this field as *leveraging human attention to data*.

1.1.3 Uncertainty management for XML

In many application scenarios of uncertainty management, information is described in a semi-structured model, because this data model provides the means to store data that lacks a rigid structure of schema. Nierman [41] states that in the types of applications where uncertainty is an issue, much of the data are not easy to represent in a relational model, even ignoring issues of uncertainty. Therefore, it is not remarkable that leading work on P-XML [41, 46, 2, 18, 29, 34, 44] all motivate the need for an uncertain semi-structured model by example of application. The flexibility of a semi-structured model and the fact that its most used representative, the eXtensible Markup Language (XML) model, is the emerging open standard for data storage and exchange over the Internet, make it attractive to investigate an extension to the XML model with uncertainty [18, 41, 34].

The above mentioned motivates an extension of the XML model with uncertainty. As a result, several data models have been introduced in research to store uncertain semi-structured data. Kimelfeld et al. [32] give an abstract view on the P-XML models of [3, 28, 29, 18, 34, 16, 45, 50]. They categorize these models in several P-XML families, which have different levels of expressive power. Document instances of a P-XML model are referred to as p-documents.

A data model goes hand in hand with a corresponding query evaluation mechanism. After all, what is the point of storing data if it cannot be used? The above mentioned P-XML models lack an efficient query evaluation mechanism. As a consequence, application scenarios of uncertainty management cannot take full advantage of P-XML models.

1.2 Research questions

We identify our problem statement as follows:

> ***There does not exist an efficient query evaluation mechanism for P-XML that is scaled up to practice.***

The main goal of this research projects is to contribute to efficient and scalable query evaluation on P-XML data. Van Keulen et al. [50] propose the following approaches to build a P-XML DBMS:

1. Instruct an XML-DBMS to cope with uncertainty.

2. Instruct an uncertain relational database management system (URDBMS) to cope with XML.

In order to contribute to efficient and scalable query evaluation on P-XML, we consider both approaches as alternative solution directions. Most research on uncertain data management focuses on relational databases [52, 35, 6, 27, 43, 9, 14]. Multiple full grown URDBMSs descend from this research that enable efficient query evaluation on uncertain relational data. We use uncertain relation technology to enable efficient query evaluation on P-XML data. Thus, we select the second approach to conduct this research. Additionally, this approach is motivated by URDBMS developers who have shown an interest towards P-XML [43].

Before we formulate our research questions, we specify a questions inherently related to our research goal.

> Q: ***Which URDBMSs are suitable to evaluate XPath queries on P-XML data and which of those is most suitable?***

We answer this question in Section 1.5.1.

We derive the following research questions from our main research goal.

> RQ1: ***How do we correctly evaluate XPath queries on P-XML as SQL queries on a URDBMS?***

We consider data in an uncertain data structure to represent a set of possible worlds. Furthermore, we consider a query specified in some query language to represent a question. We specify a P-XML into URDBMS data mapping f such that the same set of possible worlds is represented under f and we specify an XPath to SQL mapping such that the same question is asked under g. As a consequence, if we ask the same question to the same set of possible worlds, we are bound to get the same answer, however, this answer is represented differently.

We have the obligation to show that the same set of possible worlds is represented under f and that the same question is asked under g. We devote Part II of this thesis to formalize a set of data mappings and query mappings that allow the same question to be asked to different data representations. We obtain the specification of f as the sequentially composition of these data mappings. Analogously, we obtain the specification of g as the sequential composition of these query mappings.

RQ2: ***How do we efficiently map P-XML data into a URDBMS?***

RQ3: ***How do we efficiently evaluate XPath queries on P-XML data on a URDBMS?***

In Part III of this thesis, we present two designs for database mapping (f, g) where f is a P-XML into URDBMS data mapping g is an SQL to XPath mapping.

The first design is based on the specification of (f, g) –the answer of RQ2. This design uses a traditional SQL to XPath mapping and a data mapping that represents the set of possible worlds represented by a p-document as a set of U-Relations.

The second design extends g with a component of f such that the data mapping is made more efficient, but query evaluation is burdened with an extra task.

In part IV of this thesis, we present a number of optimizations for both design and conduct a performance study on both.

1.3 Global approach to show correctness

This section provides an introduction to Part II of this thesis and is intended for those interested in our specification of a correct P-XML into URDBMS data mapping and corresponding XPath to SQL mapping. This section can be skipped for those only interested in the implementation and design aspects of both mappings.

In order to specify f –a P-XML into URDBMS data mapping– and corresponding g –a XPath to SQL mapping–, we have the obligation to specify the semantics of f and g, and show that our specification conforms to these semantics. A high level illustration of the semantics of f and g is found in Figure 1.1. This figure shows a diagram constructed of nodes and edges. Nodes represent data models and edges represent mappings from one data model to another. We derive a specification for f and g from a series of mappings that are illustrated in Figure 1.2. Analogously to Figure 1.1, nodes represent data models and edges represent mappings. We discuss both figures in more detail below.

Data structures in Figure 1.1 The front view of Figure 1.1 shows two data models: P-XML and U-Rel. We consider a data model as a query language and a data structure for which query evaluation is defined. P-XML is a data model for the uncertain XML data. U-Rel is a data model for uncertain relational data. Both data models are based on the possible worlds model. This model is described in Section 2.4.1. The possible worlds model dictates databases of an uncertain data model to represent a set of possible worlds. In other words, the data structure of P-XML and U-Rel has the semantics of a set of possible worlds. This is illustrated in Figure 1.1 with the arrows sem_{pxml} and sem_{ur}.

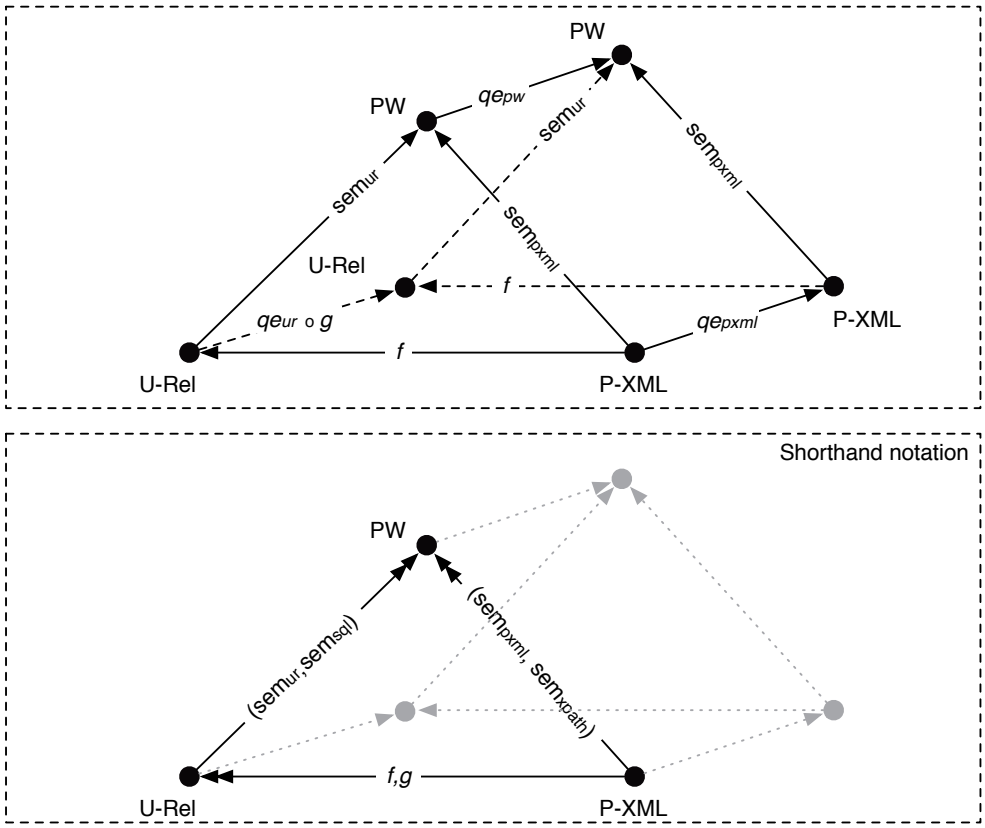


Figure 1.1: The semantics of a P-XML to U-Rel mapping

Query evaluation in Figure 1.1 The front view and rear view of Figure 1.1 are connected with different query evaluation mechanisms, denoted as qe. We consider query evaluation on a data structure as a function that takes a query and a database as input and returns the result of that query evaluated on that database. For example, the query evaluation mechanism qe_{pxml} takes a P-XML query and a p-document and returns the result of that query evaluated on that document instance such that a following query can be evaluated on the result of a preceding query. Likewise, query evaluation on PW and U-Rel return a result that conforms to the data structure on which a query is evaluated. For completeness, we note that queries for qe_{pxml} are specified in the XPath query language and queries for qe_{ur} are specified for the SQL query language.

Query results in Figure 1.1 The rear view of 1.1 denotes the data structures that derive from query evaluation. For example, the result of qe_{pxml} conforms to the P-XML data structure. Since the P-XML data structure has the semantics of a set of possible worlds, the result of qe_{pxml} has the semantics of a set of possible answers, each provided by one of the possible worlds represented by the p-document used as input. Hence, the possible worlds semantics that apply to data structures apply to query answers derived from these data structures as well.

Translation from Figure 1.1 to problem statement Our problem statement states that efficient query evaluation on P-XML –denoted with qe_{pxml}– is unknown. In our approach, we aim to evaluate P-XML queries with uncertain relational technology: we want to evaluate P-XML queries with the uncertain relational query evaluation mechanism qe_{ur} in order to bypass qe_{pxml}.

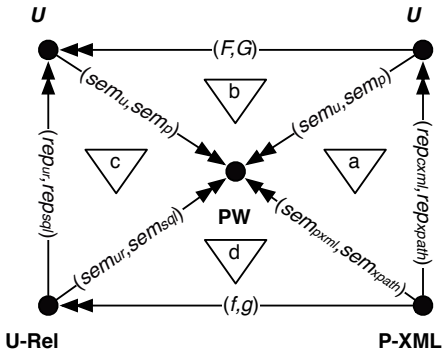


Figure 1.2: Derive the specification of f and g from a set of mappings

From problem statement to research goal We construct a P-XML to U-Rel database mapping as tuple (f, g) with a data mapping f and query mapping g. Data mapping f maps p-documents –database instances of P-XML– to U-Relations –database instances of U-Rel– such that (1) the set of possible worlds pw represented in P-XML is semantically equivalent to the set of possible worlds pw' represented in U-Rel –illustrated as the triangle (f, sem_{ur}, sem_{pxml}) that is the front view of Figure 1.1–, and (2) the semantics of the answer of a query q evaluated on pw represented by a p-document is semantically equivalent to the answer of q evaluated on pw represented by a set of U-Relations –illustrated as the triangle (f, sem_{ur}, sem_{pxml}) that is the rear view of Figure 1.1. We use g to translate the XML variant of a query q to an SQL variant of q such that $qe_{pxml}(d_{pxml}, q_{xpath}) \equiv qe_{ur}(f(d_{pxml}), g(q_{xpath}))$ where d_{pxml} is a P-XML data set[1].

How to achieve research goal Our goal is to show that our specification of f and g satisfies the above mentioned two properties such that (f, g) forms a database mapping. In order to accomplish this goal, we have to show that each side of the diagram in Figure 1.1 commutes[2]. We accomplish this with an extension of Figure 1.1 to Figure 1.2. Each double headed arrow with parameters (x, y) denotes a database mapping constructed as a data mapping x and query mapping y such that x commutes under query evaluation. We identify triangles a, b, c and d. Each of these triangles refers to the triangle constructed as the three closest nodes; triangles a, b, c and d are solely used for naming convention. The short hand notation of Figure 1.1 corresponds with triangle d that is constructed of the nodes U-Rel, P-XML and PW. Our approach to show correctness of (f, g) is as follows: (f, g) forms a database mapping $\Leftarrow$ triangle d commutes $\Leftarrow$ triangles a, b and c commute.

Idea behind this approach The extension of Figure 1.1 to Figure 1.2 is based on the following. Previous work of Antova [5] shows the construction of a URDBMS as a traditional relational database management system (RDBMS) extended with an uncertainty management mechanism such that the resulting URDBMS adheres to the possible worlds semantics. Since this mechanism is proven to extend the traditional relational data model to manage multiple states, we exploit it for other purposes: we define an abstract formalism of a traditional data model, denoted with $\mathcal{R}$, and extend it with a similar uncertainty management mechanism in order to obtain $\mathcal{U}$, an abstract formalism of an uncertain data model. Since $\mathcal{U}$ and the URDBMS of Antova share a similar uncertainty management mechanism, they integrate the possible worlds model likewise. We use $\mathcal{U}$ to show commutativity by each of the sides in Figure 1.1. We accomplish this as follows: we express P-XML in $\mathcal{U}$ and refer to the result as $\mathcal{U}_{node}$. Likewise, we express U-Rel in $\mathcal{U}$ and refer to

[1]The precise behaviour of f and g is: $f^{-1}(qe_{ur}(f(d_{pxml}), g(q_{xpath}))) = qe_{pxml}(d_{pxml}, q_{xpath})$ where d_{pxml} is a P-XML data set

[2]Commutative property of a diagram: all directed paths with the same start and end point lead to the same result by function composition.

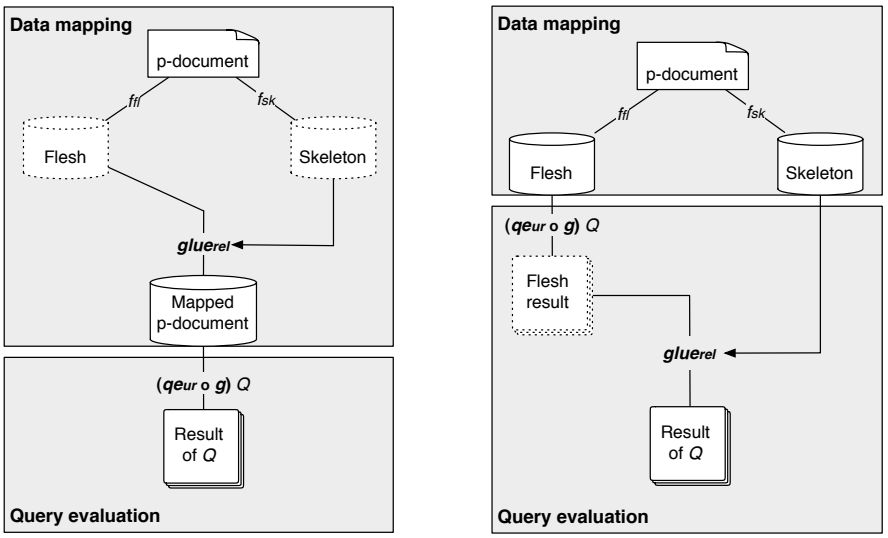
the result as $\mathcal{U}_{row}$. Since one formalism is used to express P-XML and U-Rel, a database mapping between the two provides the foundation to define a P-XML to U-Rel database mapping.

We use Figure 1.2 as leitmotif for Part II of this thesis in order to answer the research question requests for correctness of our approach.

1.4 High level design

This section provides an introduction to Part III of this thesis and is intended for those interested in our design of a P-XML into URDBMS data mapping and corresponding XPath to SQL mapping. This section can be skipped for those only interested in the high level approach to obtain a correct specification of both mappings.

We design two P-XML into URDBMS data mappings with corresponding an XPath to SQL mapping. Both designs are illustrated on a high level in Figures 1.3a and 1.3b. They give the same results for XPath evaluation on P-XML data.



(a) Document oriented gluing & t-query (b) Query result oriented gluing & tg-query

Figure 1.3: Two designs for a PXML into URDBMS mapping

First design of a P-XML to U-Rel mapping A high level illustration of our first design is found in Figure 1.3a. We first describe the design of the data mapping. A p-document is divided into flesh and skeleton. The flesh is constructed of all ordinary nodes of a p-document. The skeleton is constructed of all distributional node of a p-document. A flesh mapping (f_{fl}) maps ordinary nodes into a URDBMS. Analogously, a skeleton mapping (f_{sk}) maps distributional nodes into a URDBMS. Next, the result of f_{fl} and f_{sk} are merged together with a glue process –denoted as $glue_{rel}$. The result of $glue_{rel}$ represents the same set of worlds as the original p-document. We refer to a glue process that is applied as part of the data mapping as a document oriented (DO) glue process.

The uncertainty management mechanism of a URDBMS ensures that the result of a traditional query on uncertain data adheres to the possible worlds semantics. Hence, the design of the corresponding query mapping –denoted as g– is a traditional XPath to SQL mapping. We refer to SQL queries derived from g as t-queries

Second design of a P-XML to U-Rel mapping A high level illustration of our second design is found in Figure 1.3b. This figure shows many similarities with our first design. We construct the data mapping as a mapping of ordinary nodes f_{fl} and a mapping of distributional nodes f_{sk}. We do not design the data mapping in such a way that the results of f_{fl} and f_{sk} are merged in order to make the data mapping more efficient.

We design the query mapping as a traditional XPath to SQL mapping –denoted as g– that includes a glue process –denoted as $glue_{rel}$. We refer to SQL queries derived from this query mapping design as tg-queries and we refer to a glue process as part of the query mapping as a query result oriented (QRO) glue process.

1.5 Scope

In Section 1.3, we sketch an approach to specify a P-XML into URDBMS database mapping. In order to use this specification in practice, we propose a design for a particular URDBMS and a particular P-XML data model. We select a URDBMS in Section 1.5.1 and a P-XML data model in Section 1.5.2. Additionally, in Section 1.5.3, we select a representative subset of XPath for which we show support.

1.5.1 Suitable URDBMS for XPath processing

Most research on uncertain data management focuses on RDBMS technology. They offer a solution to store uncertain table-structured data. Examples of URDBMSs are Trio [52], MayBMS [35, 6, 27], Monte Carlo Database System (MCDB) [43], Mystiq [9], Orion [14] and ULDBs [7, 17, 4].

For scalable and efficient XPath processing, we are interested in a full grown implementation of a URDBMS. Only three candidates satisfy this criteria: Trio, MayBMS and MCDB. MCDB was not available at the start of this research and therefore, we did not consider MCDB. We consider Trio and MayBMS to be suitable URDBMSs for XPath processing.

Hollander et al. [26] made an attempt to build a P-XML database on top of Trio. Their benchmark results show that XPath queries do not scale well. Furthermore, their research identified problems with Trio managing large data sets. There is no research known that investigated XPath evaluation on P-XML data with MayBMS apart from our first attempt in previous work [48]. Based on the previous, we identify MayBMS as the most suitable URDBMS to evaluate XPath queries on P-XML data.

1.5.2 Probabilistic XML model

Multiple P-XML data models exists. In Section 2.5, we refer to the research of Kimelfeld et al. that categorizes different P-XML data models in different P-XML families based on their their expressive power. It holds that a data mapping exists from less expressive data models to more expressive data models without a data blowup. However, such a data mapping does not exists the other way around.

Earlier work [50] addresses the similarities between the uncertainty distribution of MayBMS and the P-XML model of Van Keulen et al. [50]. This P-XML model is member of the $\mathsf{PrXML}^{\{\mathsf{ind},\mathsf{mux}\}}$ family [33]. In this thesis, we specify and implement a P-XML into URDBMS mapping for the P-XML data model of Van Keulen et al.

1.5.3 XPath support

In this section, we describe a representative subset of XPath with which we conduct our research.

- Our approach is based on the schema-based mapping Shared Inlining (SI). As a consequence, only p-documents with an associated Document Type Definition (DTD) are supported[3].

[3] A DTD has to describe the flesh of a p-document

- We use a representative subset of the XPath language for which we show correctness and efficient evaluation. We define this subset as:
  - Relative location steps and absolute location steps.
  - The following XPath axes: child, descendant, descendant-or-self, ancestor-or-self, ancestor, parent, following-sibling, preceding-sibling, following, preceding, attribute, self.
  - Node tests.
  - Zero or more predicates.
  - Boolean expressions (OrExpr, AndExpr, EqualityExpr, RelationalExpr).
  - Numeric expressions (AdditiveExpr, MultiplicativeExpr, UnaryExpr).
  - Lexical structures that are also supported by PostgreSQL 8.3.3.
  - String functions that are also supported by PostgreSQL 8.3.3.

In previous work [48], we show feasibility for a P-XML into URDBMS mapping based on the schema-less XML into RDBMS mapping XPath Accelerator (XA) [19]. Unfortunately, benchmark results show undesired query evaluation behaviour: simple XPath queries evaluated on relative small data sets performed poorly. We suspected the RDBMS not to cope with the element encoding of XA. In the light of this previous research, we were motivated to use a different element encoding in order to improve query evaluation performance. This resulted in a new XML into RDBMS mapping that we use as foundation for our P-XML into URDBMS data mapping.

1.6 Contributions

The aim of this work is to present a specification for P-XML into URDBMS data mapping f and corresponding XPath to SQL mapping g such that XPath queries on P-XML data are evaluated as SQL queries on a URDBMS.

- We present multiple designs of f and g.
- We propose to evaluate one component of f as part of the query evaluation process such that the evaluation of f as well as the query evaluation process are more efficient in most scenarios.
- We validate performance of f and performance of XPath evaluation with an extensive performance study on real world data and synthetic data. Benchmark results show XPath query execution times of a few milliseconds on data sets ranging from 10^5 nodes to 10^6 nodes for a diversity of XPath expressions.

1.7 Outline

This thesis consists of four parts.

Part I — Prologue Part I consists of two chapters. Chapter 2 presents a number of topics that form the background information of this work such as an introduction to XML, RDBMS, XML into RDBMS mappings, the possible worlds model and uncertain databases. Chapter 3 presents a high level overview of our approach to specify a P-XML into URDBMS data mapping and corresponding XPath to SQL mapping with which XPath queries on P-XML data are evaluated as SQL queries on a URDBMS.

Part II — Specification Part II consists of four chapters. Chapter 4 presents $\mathcal{U}$, an abstract formalism of an uncertain data model. We define $\mathcal{U}$ as three concepts that capture a query language and a data structure for uncertain data for which query evaluation is defined. Chapter 5 presents our advancing understanding of a database mapping from P-XML to $\mathcal{U}_{node}$ –$\mathcal{U}_{node}$ is $\mathcal{U}$

that represents a tree-structure data model. Chapter 6 presents our advancing understanding of a database mapping from U-Rel to $\mathcal{U}_{row}$ –$\mathcal{U}_{row}$ is $\mathcal{U}$ that represents a table-structured data model. Chapter 7 presents our advancing understanding of a database mapping from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$.

Part III — Design Part III consists of three chapters. Chapter 8 presents a design of a P-XML into URDBMS data mapping. This design follow the specification of a P-XML into URDBMS mapping in Part II. This design is based on a dichotomy of p-documents to flesh and skeleton which are mapped into a URDBMS separately. The flesh of a p-document is constructed of solely ordinary node, the skeleton is constructed of solely distributional nodes. Chapter 9 presents the tools to merge the results of the flesh mapping and the skeleton mapping. This merging process is referred to as *gluing*. Chapter 10 presents multiple glue methods and glue method applications with which glue processes are constructed. Our first design of a P-XML into URDBMS data mapping incorporates a glue process as part of the data mapping. Our second design incorporates a glue process as part of the query mapping.

Part IV — Validation Part IV consists of two chapters. Chapter 11 presents an overview of optimizations that improve evaluation of XPath queries on P-XML data as SQL queries on a URDBMS. Based on the two designs in Part III, we built a prototype that includes the optimizations in Chapter 11. Chapter 12 presents an extensive performance study on this prototype for (1) P-XML into URDBMS data mappings and (2) XPath evaluation on a URDBMS.

Part I

Prologue

Chapter 2

Preliminaries

This chapter covers several topics that we consider as background information. Most topics –such as XML, XPath, RDBMSs, SQL– are generally known in the field of databases. We also describe less familiar topics that are related to this research. These topics include the XA approach and the SI approach –two XML into RDBMS mappings–, an introduction to the possible worlds model and uncertain databases.

2.1 An abstract view on database mappings

Database mappings allow queries specified in one data model to be evaluated by the query evaluation mechanism of another data model. A database mapping is constructed of a data mapping with a corresponding query mapping. A data mapping maps content of one data structure to another data structure. A data mapping solely provides an approach to store the same data in a different representation. In order to take advantage of such a representation, a query mapping is required that allows a question specified in one language to be asked in another language such that the same questions can be asked to different data representations. A query mapping g corresponds with a data mapping f if the result of a query q evaluated on one data structure db is similar to the result of another query q' evaluated on another data structure db' such that $q' = g(q)$ and $db' = f(db)$ for each db and q.

Figure 2.1 provides an abstract view on database mappings for a data mapping f with corresponding query mapping g. If we apply data mapping f to db, we retrieve db'. Likewise, if we apply data mapping f to ans, the result of a query evaluated on db, we retrieve ans'. Queries are evaluated with a query evaluation mechanism, denoted with qe. The diagram in Figure 2.1 has the commutative property, which means that:

$$f(qe(db, q)) = qe'(f\ db, g\ q)$$

We define a database mapping as the double (f, g) that satisfies the commutative property.

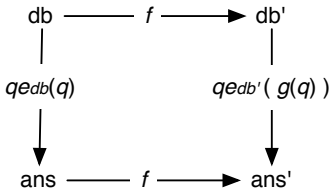


Figure 2.1: Visualization of a database mapping

2.2 Introduction to XML

2.2.1 Extensible Markup Language

XML is a semi-structured data model that represents information as a tree. In this section, we specify XML with a schema. Therefore, we first postulate a collection of nodes and a collection of text:

$$[NODE, TEXT]$$

Schema *ABS-XML* defines XML-related data structures as follows:

ABS-XML

$$
\begin{array}{l}
rootnode : NODE \\
xmlnodes, textnodes, ordnodes : \mathbb{P}\ NODE \\
edge, \texttt{/parent} : NODE \nrightarrow NODE \\
\texttt{/child} : NODE \leftrightarrow NODE \\
\texttt{/ancestor}, \texttt{/ancestor-or-self} : NODE \leftrightarrow NODE \\
\texttt{/descendant}, \texttt{/descendant-or-self} : NODE \leftrightarrow NODE \\
getTag : NODE \nrightarrow TEXT \\
getPCData : NODE \nrightarrow TEXT \\
\hline
\langle XMLnodes, textnodes \rangle \ \mathsf{partition}\ ordnodes \\
\mathrm{ran}\ edge \cap textnodes = \varnothing \\
\texttt{/parent} = edge \\
\texttt{/ancestor} = edge^{+} \\
\texttt{/ancestor-or-self} = edge^{*} \\
\texttt{/child} = edge^{\sim} \\
\texttt{/descendant} = (edge^{\sim})^{+} \\
\texttt{/descendant-or-self} = (edge^{\sim})^{*} \\
\mathrm{dom}\ getTag = XMLnodes \\
\mathrm{dom}\ getPCData = textnodes
\end{array}
$$

XML

$$
\begin{array}{l}
ABS\text{-}XML \\
\hline
rootnode \in XMLnodes \\
\mathrm{dom}\ edge = ordnodes \setminus \{rootnode\}
\end{array}
$$

Schema XML specifies document instances of XML as a tree. Edges represent the child/parent-relationship between nodes in the p-document. The XML-schema distinguishes two node kinds: XML nodes *xmlnodes* and text nodes *textnodes*. Nodes of one of these two node kinds are referred to as ordinary nodes, denoted as *ordnodes*. The function *getTag* is defined for the former and returns the tag of an XML node. Nodes that have the same tag are of the same element type. The function *getPCData* is defined for the latter and returns the PCData value of text nodes.

We capture most XPath axes as a mutual relation between nodes based on *edges*. The semantics of these axes are found in Table 2.1. We highlight: the child axis is the inverse of the parent axis, the ancestor axis is the transitive closure of the parent axis and the ancestor-or-self axis is the reflexive transitive closure of the parent axis.

Exterior to the XML-schema, we define a path as a sequence of nodes such that from each of its nodes there exists an edge to the next node in the sequence. Any path between two nodes is unique in a tree. We denote a path from a node n to a node m as $\uparrow_{n,m}$. We write $n \in \uparrow_{m,l}$ to state that node n lies on path $\uparrow_{m,l}$. We write $\uparrow_n$ to denote a path from n to the root of a document (including p-documents and possible documents). All nodes that lie on $\uparrow_n$ define the ancestor-or-self axis of n.

2.2.2 XPath expressions

Tree-traversals in XML-documents are specified in XPath [8]. We give a simplified XPath syntax:

$$
\begin{array}{l}
\texttt{XPATH} ::= \text{`/'}, (\texttt{step}, \text{`/'})^{*} \\
\texttt{step} ::= \texttt{axis} :: \texttt{nodetest}[\texttt{pred}]^{*} \\
\texttt{axis} ::= \texttt{/parent} \mid \texttt{/child} \mid \ldots \\
\texttt{nodetest} ::= name \mid * \\
\texttt{pred} ::= \text{`.'}, \texttt{XPATH} \mid \texttt{XPATH} \mid bool\_expression
\end{array}
$$

Axis α	Result
`self`	v
`child`	child nodes of node v
`descendant`	recursive closure of `child`
`descendant-or-self`	union of `child` and `descendant`
`parent`	parent node of v
`ancestor`	recursive closure of `parent`
`ancestor-or-self`	union of `ancestor` and `self`
`following`	nodes following v in document order
`preceding`	nodes preceding v in document order
`following-sibling`	`following` with same parent as v
`preceding-sibling`	`preceding` with same parent as v
`attribute`	attribute nodes of v
`namespace`	namespace nodes of v

Table 2.1: Semantics of axis α supported by XPath (step v/α).
(table is borrowed from [19])

XPath expressions specify tree-traversals via two parameters: (1) a context node, which is the starting point of the tree-traversal, and (2) a sequence of location steps (`step`) syntactically separated with a `/`-sign. Each location step takes a set of context nodes as input and returns a set of nodes that serve as context nodes on which the following location step is evaluated. The result of the last-mentioned location step is the result of the tree-traversal. Location steps are of the form `/axis` :: `nodetest`[`predicate`]* where (1) `axis` is one of the listed axes in Table 2.1, (2) `nodetest` specifies a node test that is either a filter that restricts the result of a location step to contain solely nodes of the element type specified by *name* or no filter in case of the $*$-symbol is used, and (3) `predicate` constrains the result of a location step with a test to which the output of a location step has to conform to. The test itself is defined as either (1) an XPath expression preceded with a dot-symbol that denotes the XPath expression to process nodes that satisfy the node test, (2) an XPath expression that starts at the root, or (3) a Boolean expression that has to evaluate to true.

2.3 XML into RDBMS mappings

In essence, XML into RDBMS mappings make an attempt to evaluate XPath queries on XML data as SQL queries on relational data. Relational databases are optimized for querying on large amounts of table-structured data. Hence, query evaluation on XML contents stored in an RDBMS can profit from RDBMS technology if a corresponding XPath to SQL query mapping is defined. We presume the XML into RDBMS mappings referred to in this section to comply with the commutative property we discussed in Section 2.1.

XML into RDBMS mappings can be categorized as *schema-based* or *schema-less*. The former requires a schema that describes the structure of a family of XML-documents, the latter does not. A schema can be of two types: a DTD or an XML Schema. In case the structure of an XML-document is conform to the defined structure of a schema, we say the XML-document is *valid*. Hence, a schema-based mapping can process XML-document in case it knows the associated schema. In contrast, a schema-less mapping can process any XML-document.

We describe the mappings Shared Inlining (SI), a representative of the schema-based approach, and XPath Accelerator (XA), a representative of the schema-less approach in Sections 2.3.1 and 2.3.2.

2.3.1 Shared Inlining, a schema-based mapping of XML to relational databases

Shared Inlining (SI) [47] is a schema-based XML into RDBMS mapping that makes use of a DTD. The basic principle behind SI is that an XML-document is assumed to contain many XML nodes of the same element type. This makes it attractive to store these nodes together in the same table as illustrated in Figure 2.2. The danger in this approach is that if an XML-document contains many element types, many tables will be created. The idea behind inlining is that some nodes don't require a separate table, but can share a table. As an additional advantage, the evaluation of the parent/child axis is free of execution costs in case parent and child nodes reside in the same table.

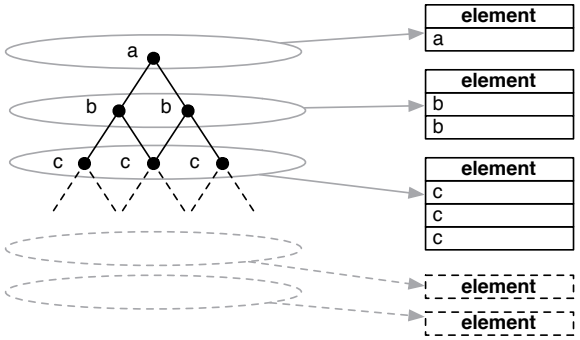


Figure 2.2: Abstract view on SI

We describe SI slightly different than the work it is introduced in [47]. We consider SI to be defined in three parts: (1) a relational storage structure to which an input XML-document is mapped, (2) an inlining principle, and (3) an encoding for the XML data structure. For reuse purposes, we define the abstract data structure Abstract Shared Inlining (ASI[*ENC*]) as the first two parts of SI and a parameter *ENC* –an encoding for the XML data structure. In Section 8.4.2, we introduce a new XML into RDBMS approach that is based on ASI[*ENC*].

Relational data structure We consider an XML-document to contain nodes of various element types:

$$\frac{N_1, \ldots, N_n : \mathbb{P}\ NODE}{\langle N_1, \ldots, N_n \rangle \text{ partition } allnodes}$$

Symbol N_i denotes a set of nodes that are of the same element type. The element type of an XML node is defined by its tag. For example, `<car/>` denotes an XML node of the element type *car*. Text nodes are considered to have an element type as well that is related to their parent. For example, "Mustang" in `<car>Mustang</car>` is considered to be a text node of element type *car.pcdata* [1].

The principle behind SI is that each element type involved in a mapping should have its own table such that nodes of a specific element type are stored at the same location. We refer to the table to which nodes of a specific element type N are mapped as the *element table of* N. We consider the function $getT()$ to return the table that is assigned to a specific element type. If we ignore the inline principle, $getT()$ defines a one-to-one correspondence between element tables and element types. However, the inline principle reduces the number of element tables if the structure of an XML-document satisfies certain inline rules. This is illustrated in Figure 2.3. The main benefit of the SI data-structure is that element types are efficiently found during query evaluation.

[1] In the original work of Shared Inlining [47], text nodes are not taken into account. Instead, "mustang" in `<car>Mustang</car>` is considered to be the value of the "car"-element.

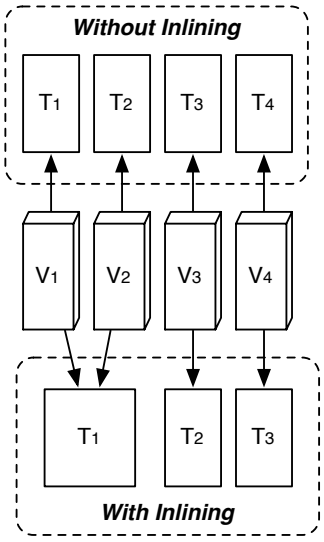


Figure 2.3: Relational data structure of SI

Inlining principle The inlining principle of Shanmugasundaram [47] defines nodes of different element types to be stored in the same table if they satisfy certain requirements specified by the set of *inline rules* defined in [47]. The advantages of the inline principle are efficient storage and no execution costs for the evaluation of the parent/child axis that involve element types that reside in one table. In order to take advantage of the inlining principle, a schema is required to validate these rules for element types that share the parent/child-relation is that a data structure needs to be created in advance which takes inlining into account.

Let N and N' be two different element types. The first inline rule specifies that nodes of N and N' share the parent/child-relationship with a multiplicity of at most 1. In other words, each node in N has at most one child node of element type N'. The second inline rule states that element type N must have an indegree of 1 in the DTD. In other words, all nodes in N' have a node in N as their parent.

In order to make inlining more intuitive, we explain inlining as element types that share the guest/host-relationship. This relationship is defined as: a host should always be present in order for a guest to visit. The host/guest-relationship is an analogy for the parent/child-relation between element types that satisfy the traditional inlining rules. In the light of relational tables, multiple nodes are stored in one row if a node of the host element type is present. We illustrate this relationship with an example.

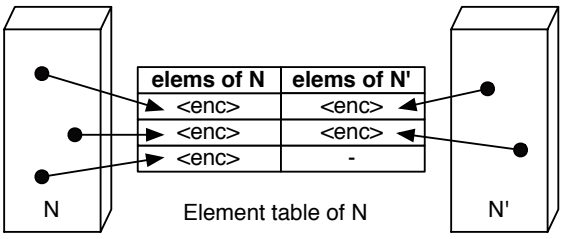


Figure 2.4: Visualization of the inlining principle

Presume two element types N and N' where $N \neq N'$ and $getT(N) = getT(N') = T$. This scenario illustrates inlining, since nodes of different types are mapped into the same table. If we

say *"N' is inlined with N"*, we define N to serve as host and N' to serve as guest. In order for a node of element type N' to be represented in a row of T, a node of element type N should be represented in the same row as well. This principle is illustrated in Figure 2.4. On the left, we have all nodes of element type N; on the right, we have all nodes of element type N'. If we inline N' with N, we store nodes of both element types in the element table of N that is illustrated in the middle.

To elaborate on inlining, a host is allowed to have multiple host/guest-relationships with different element types, and a guest element type may also take the role of host to invite other element types to start a guest/host-relationship with. Note that there is a subtle difference between the two. We illustrate this difference based on our previous example.

We introduce element type N'' unequal to N and N'. In case N'' has the guest/host-relationship with N' and a node in N'' is represented in a row, then a node in N' and a node in N also have to be represented in the same row. This requirement is not in effect if N'' has a guest/host-relationship with N, since only a node in N has to be represented in the same row.

It can be proven that whatever element types have the guest/host-relationship with one another in an element table, there is exactly one host element type that is not guest of any other element type. We refer to this element type as the *master element type* of its element table. All other element types that reside in the same element table as the master of that element table are *slave element types*. In case of Figure 2.4, element type N is the master of the element table depicted in the middle.

Encoding for XML data SI represents nodes as follows:

- Nodes of an element type that serve as master and do not have a text node as child are represented as $\langle$`elemID` : *integer*, `parentCODE` : *integer*, `parentID` : *integer*, `isroot` : *boolean*$\rangle$.
- Nodes of an element type that serve as slave and do not have a text node as child are represented as $\langle$`elemID` : *integer*$\rangle$.
- Nodes of an element type that serve as master and have a text node as child are represented as $\langle$`elemID` : *integer*, `pcdata` : *text*$\rangle$.
- Nodes of an element type that serve as slave and have a text node as child are represented with the PCData-value of their text node child $\langle$`pcdata` : *text*$\rangle$.

In this encoding, (1) `elemID` : *integer* represents a unique identifier for XML nodes, (2) `parentCODE` : *integer* refers to the element type of the parent, (3) `parentID` : *integer* provides a pointer to the parent node, (4) `isroot` : *boolean* is used to identify the root of an XML-document, and (5) `pcdata` : *text* maps PCData.

2.3.2 XPath Accelerator, a schema-less mapping of XML to relational databases

XPath Accelerator (XA) [19, 23, 22, 20, 21] is an encoding for XML-data that has been specifically designed to support the evaluation of XPath queries with relational database technology. The XA approach is known under slightly different element encodings. The latest work on XA [21] studied the application of the XA approach to off-the-shelf RDBMSs without a change of the underlying database kernel. In this section, we provide a summary of this work and consider each reference to XA to point to this work specifically.

Data encoding The XA approach uses the *pre-order* rank, *size* and *level* of XML nodes to map the tree-structure of XML-documents onto a table-structure.

- Pre-order ranks are retrieved using a specific *document traversal*: a counter-clockwise traversal of the perimeter of an XML-tree starting at the root of the tree. Pre-order ranks are assigned in the order in which document nodes are discovered during a document traversal. An illustration of this document traversal is found in Figure 2.5.

- The size of an XML node is defined as the number of nodes of the subtree rooted at that XML node excluding the XML node itself.
- The level of an XML node is defined as the number of nodes that reside in the ancestor axis of that XML node.

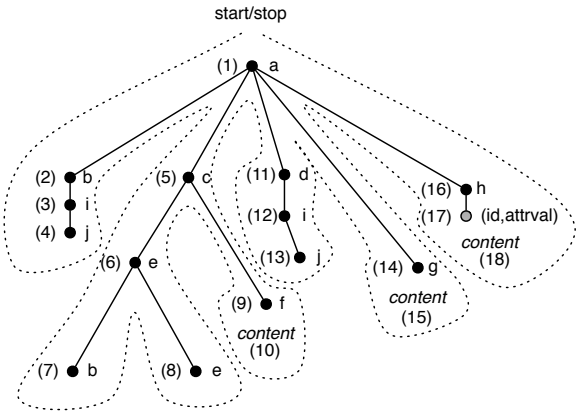


Figure 2.5: Counter-clockwise traversal of the perimeter of an XML-document starting at the root

To support further XPath axes and node tests, the mapping of a node n is accomplished as:

$$n \mapsto \langle pre\ n, size\ n, level\ n, tag\ n, kind\ n \rangle$$

where the tag-name *tag* of n saves the element type of n , and the type of n is associated with the node kind of n. A node kind can be of the types: `XML node` (`elem`), `XML-attribute` (`attr`), `PCData`. The mapping of Figure 2.5 onto a table-structure is found in Figure 2.6. PCData values and attribute values are stored in the right table 'Contents'. All other information is stored in the left table 'Accel'. Table 'Contents' has foreign keys to entries in 'Accel' such that text nodes are associated with PCData and attributes are associated with associated attribute values.

To make this encoding more intuitive, we represent each node as square boxes in a two-dimensional plane, illustrated in Figure 2.7. The order of identifiers corresponds with the document order of nodes. The embedded nature of boxes corresponds with the embedded nature of nodes.

Query Evaluation XA [21] has been specifically designed to support the evaluation of XPath axes with relational database technology. The following axes are evaluated by an RDBMS with a combination of simple range conditions and equality conditions. Let c be a context node and n be an arbitrary node in an XML-document:

$$\begin{array}{lcl}
n \in c\texttt{/descendant} & \Leftrightarrow & pre(c) < pre(n) \leq pre(c) + size(c) \\
n \in c\texttt{/descendant-or-self} & \Leftrightarrow & pre(c) \leq pre(n) \leq pre(c) + size(c) \\
n \in c\texttt{/child} & \Leftrightarrow & pre(c) < pre(n) \leq pre(c) + size(c) \wedge level(n) = level(c) + 1 \\
n \in c\texttt{/ancestor} & \Leftrightarrow & pre(n) < pre(c) \leq pre(n) + size(n) \\
n \in c\texttt{/ancestor-or-self} & \Leftrightarrow & pre(n) \leq pre(c) \leq pre(n) + size(n) \\
n \in c\texttt{/parent} & \Leftrightarrow & pre(n) < pre(c) \leq pre(n) + size(n) \wedge level(c) = level(n) + 1 \\
n \in c\texttt{/preceding} & \Leftrightarrow & pre(n) + size(n) < pre(c) \\
n \in c\texttt{/following} & \Leftrightarrow & pre(c) + size(c) < pre(n) \\
n \in c\texttt{/self} & \Leftrightarrow & pre(c) = pre(n)
\end{array}$$

Notice that the definition of the symmetry of XPath axes is also reflected by these rules.

Figure 2.7 provides insight in how the ancestor/descendant axis and the following/preceding axis follow from the XA encoding; just apply the previous definitions to the square boxes to determine which axes apply to what boxes and observe that the same relations follow from the embedded nature of the boxes.

pre	level	size	tag	kind
1	1	16	'a'	elem
2	2	2	'b'	elem
3	3	1	'i'	elem
4	4	0	'j'	elem
5	2	5	'c'	elem
6	3	2	'e'	elem
7	4	0	'b'	elem
8	4	0	'e'	elem
9	3	1	'f'	elem
10	4	0	-	pcdata
11	2	2	'd'	elem
12	3	1	'i'	elem
13	4	0	'j'	elem
14	2	1	'g'	elem
15	3	0	-	pcdata
16	2	1	'h'	pcdata
17	3	0	'id'	attr
18	3	0	-	pcdata

Table ***Accel***

pre	text
10	*content*
15	*content*
17	*attrval*
18	*content*

Table ***Contents***

Figure 2.6: The result of the XA approach applied to Figure 2.5

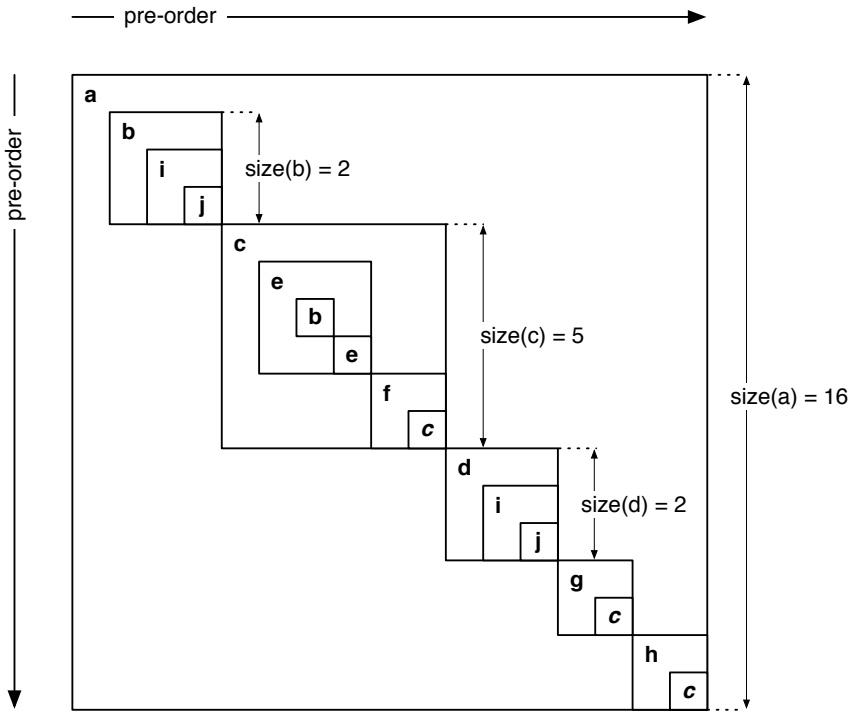


Figure 2.7: Two dimensional plane of the (pre,size)-encoding

2.4 Introduction to uncertain databases

We introduce the concept of an uncertain database as

2.4.1 Interpretation of an uncertain database in terms of possible worlds

The concept of an uncertain database originates from the *possible worlds model*. This model is founded on the fact that uncertainty in data makes it impossible to define what precisely the real world is. However, there is a notion of what *possible worlds* exist given the available information at hand, and the actual world is considered to be one of them. A single possible world is considered to be a set of objects. An object may be part of multiple possible worlds. The set of possible worlds of which an object is member is called the *world set* of that object.

An uncertain database is designed to store a compact representation of a set of possible worlds. Each possible world is represented as a complete state of the database. Since an ordinary database has solely one state, it can be viewed as a single possible world [25]. In order to represent possible worlds, database objects are annotated with information that describes the world set of that object. We use the term uncertainty distribution to refer to this kind of information. The uncertainty distribution of an uncertain database captures how database objects are randomly selected in an uncertain database.

Important to note: in theory, queries that apply to traditional databases also apply to uncertain databases: a traditional query executed on a traditional database results in one (certain) answer; a traditional query evaluated on an uncertain database results in a set of possible answers, each provided by one of the possible worlds.

2.4.2 Granularity of uncertainty

We introduce the terminology Unit of Choice (UoC) to refer to the granularity of uncertainty in an uncertain database. We use a relational database to illustrate this concept: a relational database stores table-structured data, which we describe as a grid data structure consisting of rows and columns. A URDBMS can be designed to support rows as unit of choice, by which multiple possible presentations of rows could be stored. Another possibility is to design a URDBMS to support columns as unit of choice. As a consequence, multiple presentations of columns can be stored. A third option is the use of row-attributes as unit of choice, which we define as the intersection of rows and columns. A URDBMS adopting a granularity of uncertainty at the attribute level stores rows with different attribute representations.

Consider the following tuples (the sign '|' should be read as an OR-separator to define possibilities):

(*Barack Obama,* {*Emperor* | *President*})
(*Barack Obama, Emperor*) | (*Barack Obama, President*)

Both tuples represent the same two possible presentations. However, they differ in unit of choice, since the former tuple supports attributes to have multiple representations while the latter allows tuples to have multiple representations. The question which of the two representations is considered to be correct cannot be answered with the available information at hand. We consider this ignorance as uncertainty.

2.5 Probabilistic XML, an uncertain semi-structured data model

Research introduced several data models to store uncertain semi-structured data. Kimelfeld et al. [32] give an abstract view on the P-XML models of [3, 28, 29, 18, 34, 16, 45, 50]. They categorize these models in several probabilistic XML families, which have different levels of expressive power.

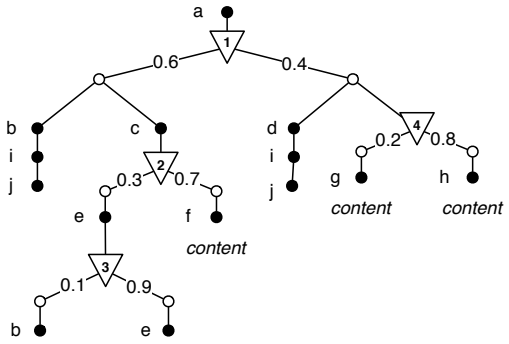


Figure 2.8: Probabilistic XML tree

All probabilistic XML models introduce new node kinds to enrich the traditional XML encoding. These node kinds are referred to as *distributional nodes*; fictive nodes that specify how their children are randomly selected in a p-document [33, 46]. Distributional nodes annotate their descendants with knowledge of which possible worlds they are member. Distributional nodes come in four different flavors: *ind*-type, *mux*-type, *exp*-type, *cie*-type. These flavors together make it possible to express any document expressed in one of the known P-XML data models. Note that the individual P-XML models may use different terminology to refer to these four distributional node kinds.

In future chapters, distributional nodes refer to possibility nodes and probability nodes that we introduce in the next section.

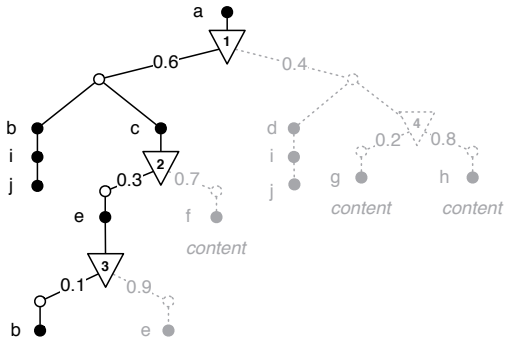
2.5.1 P-XML model of Van Keulen et al.

In this research, we use the P-XML data model of Van Keulen et al. [50] – member of the $\mathsf{PrXML}^{\{\mathsf{ind},\mathsf{mux}\}}$ family [33] – as foundation for this research. This data model is based on two kinds of *distributional nodes*: (1) *probability nodes* ($\triangledown$) to represent different choice points; places in an XML-document where choices have to be made, and (2) *possibility nodes* ($\circ$) to represent alternatives of which a choice point may select its pick. We refer to the selection of an alternative by a choice point as the *choice* of that choice point. The likeliness of an alternative to be selected is captured with a probability p that satisfy $0 < p \leq 1$. Choices are mutually exclusive; the sum of the probabilities of all choices associated with one choice point is equal to 1.

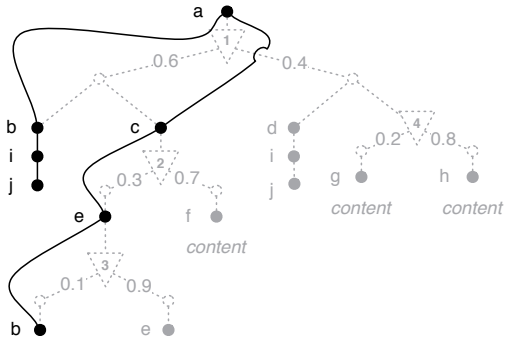
An example p-document is found in Figure 2.8. We use this example as a running example in future chapters. This example has four choice points to which we refer as $v_1, \ldots, v_4$. Choice points restrict the membership of their descendants to be part of a possible world. For example, node $n_{tag=d}$ has one ancestor choice point v_1. In order for $n_{tag=d}$ to be part of a possible world, the second possibility child of v_1 with probability 0.4 has to be selected. We refer to possibility nodes as $d_{i,j}$ where i refers to its parent choice point v_i and j to the nth alternative of v_i counted from left to right. Node $n_{tag=h}$ has two ancestor choice points v_1 and v_4. Both have to choose their right alternative $d_{1,2}$ and $d_{4,2}$ in order for $n_{tag=h}$ to be member of a possible world. If we assume independence of the selections of $d_{1,2}$ and $d_{4,2}$, the likelihood for $n_{tag=h}$ to be part of a possible world is equal to the product of both possibility nodes to be selected: $P(d_{1,2}) \cdot P(d_{4,2}) = 0.4 \cdot 0.8 = 0.32$.

A p-document represents a set of possible worlds. In order to retrieve a possible world, one alternative has to be selected for each choice point by which all unselected alternatives are discarded. The result is a *possible document*; a p-document that represents exactly one possible world where each probability node has exactly one child possibility node. A *random document* is distilled out of a possible document by removing all distributional nodes whereby children of selected alternatives are connected to the closest non-distributional node. Since distributional nodes are removed in the retrieving process of a random document, they are considered to be fictive nodes.

Figure 2.9a illustrates one possible document represented by the p-document in Figure 2.8. In



(a) Single possible world representation



(b) Random document

Figure 2.9: Possible world extraction

this example, the left alternative of each choice point is selected. As the figure illustrates, the choice to select the left alternative of choice point v_4 does not change the resulting possible document. The random document associated with this possible document is found in Figure 2.9b.

The set of possible worlds of which a node is member is referred to as the *world set* of that node. A world set is described with a set of choices or alternative selections. We refer to the set of choices that identify the world set of a node as its world set descriptor (WSD).

2.5.2 Query evaluation on P-XML data

Query evaluation on P-XML is discussed in the work of Van Keulen et al. [51]. They derive query evaluation from the possible worlds model. This model is the foundation for uncertain databases and is discussed in Section 2.4.1. In essence, an uncertain database represents a set of possible worlds. Each possible world corresponds with one state of the database. The result of a query evaluated on a set of possible worlds is a set of possible answers, each provided by one of the possible worlds separately. An uncertain database based on P-XML has to adhere to the possible worlds semantics. Therefore, one inefficient approach to support query evaluation on p-document is to evaluate an XML-query on each possible world represented by that p-document separately and merge the results into a set of possible answers.

2.6 Views, unmaterialized and materialized

A view is a virtual table representing a select-query. A view is either materialized or unmaterialized. An unmaterialized view is a gateway to a query; a use of an unmaterialized view requires the

evaluation of the query associated with the used unmaterialized view. In contrast, a materialized view caches the result of a query, but behaves like a view. That is, the data in a materialized view changes when the data in the underlying data structures changes from which the materialized view is constructed.

Chapter 3

From P-XML to U-Rel, the Detour

In this chapter, we give a high level overview of our approach to show correctness for database mapping (f, g) that maps P-XML to U-Rel. A data mapping transforms data from one representation to another such that the same information is preserved. Additionally, a query mapping allows the same question to be asked to a different data representation. Hence, if two data representations represent the same information, asking the same question result in the same answer, however, the representation of that answer differs.

Our approach to show that our design of (f, g) is correct is based on four database mappings that map P-XML data to three intermediate data models. For each of the four database mappings (f', g'), we have to prove that the same set of possible worlds is represented under f' and the same questions are asked under g'. This implies that the commutative property holds for each (f', g'). If all four database mappings are sequentially composed, the result is a P-XML to U-Rel database mapping (f, g). This approach is illustrated in Figure 3.1.

We describe each of the four database mappings in one of the following sections. In Chapters 4, 5, 6 and 7 we attempt to show correctness of these mappings.

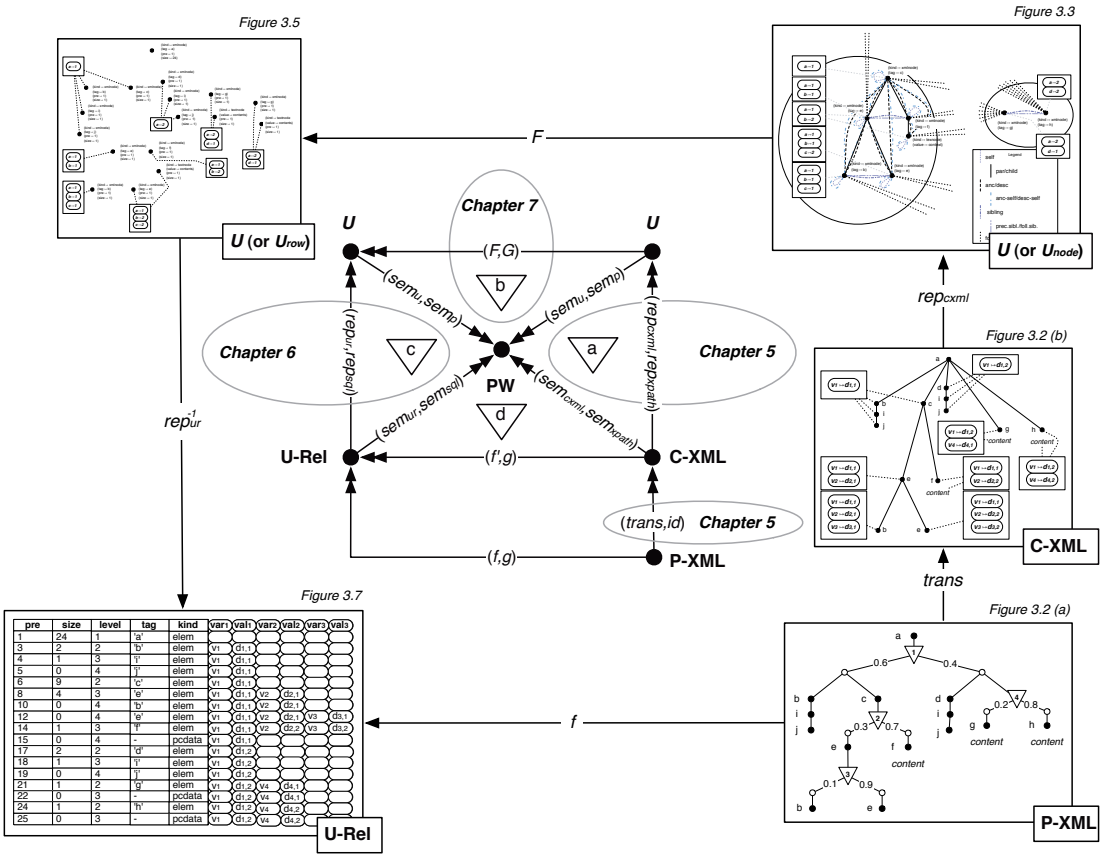


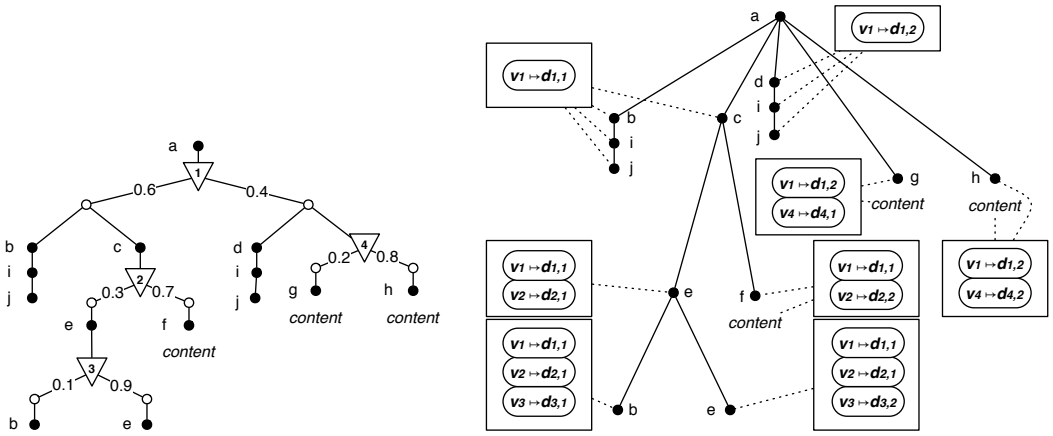
Figure 3.1: From P-XML to U-Rel, the detour

3.1 From P-XML to C-XML

We map data from P-XML to C-XML in order to disconnect the representation of uncertainty from the tree-structure. We accomplish this by replacing distributional nodes with node annotations. We refer to these node annotations as choice point assignments (CPAs). CPAs describe the world set of nodes.

Data mapping In order to map a P-XML document instance –called a p-document– to a C-XML document instance –called a c-document–, we introduce data mapping *trans*. Mapping *trans* consists of a number of steps. In the first step, distributional nodes are extracted from the input p-document such that we obtain (1) a set of distributional nodes *ds*, and (2) an ordinary XML-document *fl* that is obtained from the remaining set of ordinary nodes. In the second step, we create a set of CPAs *sk* from *ds*. More specifically, each pair (v, d) of child possibility node d and parent probability node v in *ds* is represented as one CPA in *sk*. In the third and final step, each ordinary node in *fl* is associated with a subset of *sk*. For each node, we require that the same world set is described under *trans*.

For illustration purposes, Figure 3.2b shows the c-document that is the result of *trans* applied to the p-document in Figure 3.2a. We designed *trans* such that for each node n in Figure 3.2a and counterpart $trans(n)$ in Figure 3.2b holds that each alternative d that lies on $\uparrow_n$ is represented as the CPA that selects d.



(a) p-document also shown in Figure 2.8 (b) c-document

Figure 3.2: A p-document and its C-XML counterpart

Query mapping Query evaluation on P-XML data is discussed in the work of Van Keulen et al. [51]. They derive the semantics for query evaluation on P-XML data from the possible worlds model and the semantics of query evaluation on the XML model such that the P-XML model and the XML model support the same query language. The same approach is applicable to the C-XML model. It follows that the same query languages are supported by the P-XML data model and the C-XML data model, however, the mechanism to evaluate these queries differs. If a p-document and a c-document encode the same information –a set of XML-documents–, no query mapping is required.

3.2 From C-XML to $\mathcal{U}_{node}$

We map data from C-XML to $\mathcal{U}_{node}$ in order to represent XPath axes as pairs of nodes. In P-XML and C-XML, the parent/child axis is represented as edges between nodes. Other XPath axes are derived from this axis. In $\mathcal{U}_{node}$, we represent each single XPath axis as edges between nodes. Each edge kind corresponds with one XPath axis. Additionally, we map CPAs to random variable assignments (RVAs) in order to express the uncertainty distribution in similar fashion as U-Rel. CPAs describe a set of possible worlds as a set of choices represented as possibility nodes assigned to probability nodes. RVAs describe a set of possible worlds as a set of choices represented as random variable assignments. A mapping of CPAs to RVAs causes possible worlds to be described in a different notation.

We write $\mathcal{U}_{node}$ instead of $\mathcal{U}$ to state that $\mathcal{U}$ represents a tree-structured data model. Analogously, $\mathcal{U}_{row}$ represents a table structured data model.

Data mapping In Chapter 4, we introduce a formalism of an uncertain data model $\mathcal{U}$. Model $\mathcal{U}$ is a graph-structured data model. We designed $\mathcal{U}$ such that table-structured data models and tree-structured data models can be represented as an instance of $\mathcal{U}$. Data entries or nodes in $\mathcal{U}$ are called objects. Objects are annotated with a world set descriptor specified with a set of RVAs. $\mathcal{U}$ uses edges to represent relations between objects and annotations to represent properties of objects. Queries are also specified as a graph structure; edges represent required relations between wanted objects and annotations represent required properties of these wanted objects.

First, we explain how we intend to represent the same set of possible worlds under *rep*, a data mapping from C-XML to $\mathcal{U}_{node}$. C-XML uses CPAs to describe the world set of nodes. Analogously, $\mathcal{U}$ uses RVAs to describe the world set of objects. Our approach represents nodes as objects and represents CPAs as RVAs. We accomplish the former with a bijection of nodes to objects. For the latter, we first discuss their similarities. For CPAs and RVAs, their mechanism to describe world sets is almost identical. The only difference is that the definition of a CPA specifies a pair of a probability node and a possibility node while the definition of an RVA specifies a pair of a random variable and an assignment value. In both definitions, the first argument of the pair represents a choice point and the second argument represents an alternative to be selected. Hence, we map world set descriptors with (1) a bijection of probability nodes to random variables, and (2) a bijection of possibility node to assignment values. For each node n in a c-document and its counterpart $rep(n)$, we have to prove that the world set of n is the same as the world set of $rep(n)$.

Second, we have to ensure that the tree-structure is preserved under $rep(n)$. We consider the tree-structure to be captured if for each node, its relative location to all other nodes in a tree is known. We accomplish this as follows. Let n, n' be two nodes and $rep(n), rep(n')$ be their counterparts. For α varied over all XPath axes, if $n' \in n/\alpha$, we relate object $rep(n)$ to object $rep(n')$ with a relation α. As a consequence, for each two nodes, their relative location to each other in a tree is known.

For illustration purposes, Figure 3.3 shows a partial mapping of the c-document in Figure 3.2b to $\mathcal{U}_{node}$. Observe a one-to-one correspondence between nodes and objects and between CPAs and RVAs. The legend in the bottom right corner shows the correspondence between XPath axes and kinds of edges.

Query mapping XML queries typically specify patterns of selection predicates on multiple elements that have some specified tree-relationships. Bruno et al. [10] propose to represent XML-queries as twig patterns. A *twig pattern* specifies a required tree-pattern or tree-structure as a twig. A twig is a small tree that uses node-labels to specify desired element types and edges to specify desired parent-child relations or desired ancestor-child relations. The concept of a twig pattern is a widely used concept in research areas that focus on efficient XML query evaluation [37, 11].

The query model of $\mathcal{U}$ and the concept of the well known twig pattern share the same semantics. However, the expressiveness of the two differs: the query model of $\mathcal{U}$ supports all XPath axes while the concept of a twig pattern supports only the parent/child axis and the ancestor/descendant

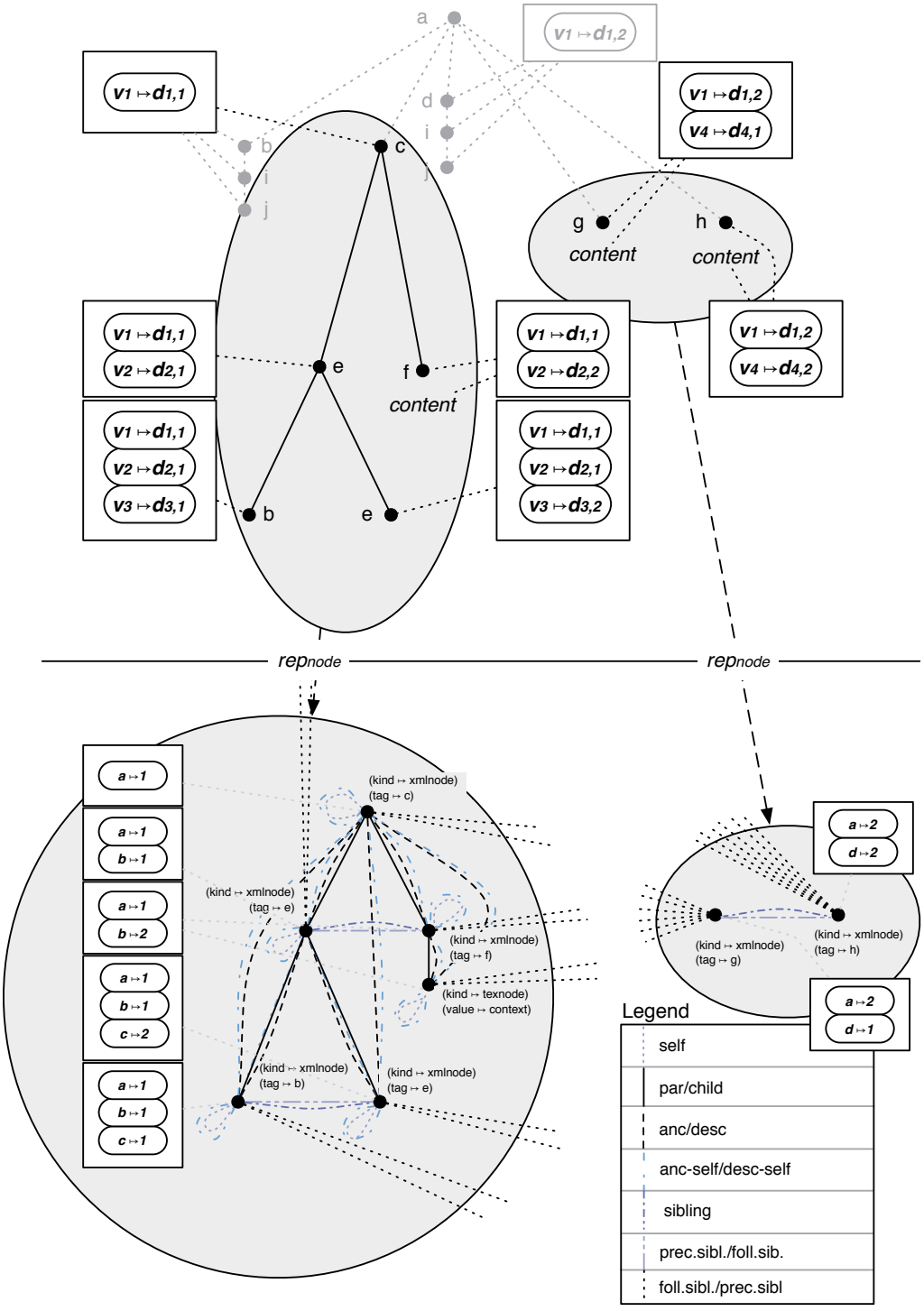


Figure 3.3: Partial mapping of C-XML to $\mathcal{U}_{node}$

axis. We refer to query instances of $\mathcal{U}$ as patterns (not twig patterns). We consider a translation of XPath to patterns to be the well known translation of XPath to twig patterns.

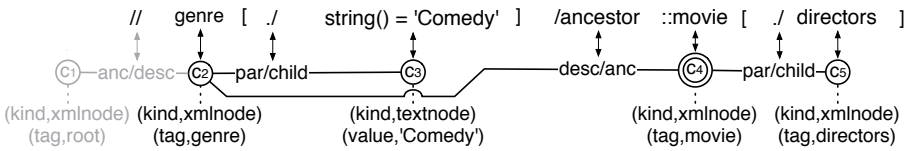


Figure 3.4: A translation of an XPath expression to a Pattern expression

For illustration purposes, Figure 3.4 shows a translation of an XPath expression to a pattern. We refer to the nodes in a pattern as candidates. We observe that node tests correspond with candidates and axes correspond with required relations between candidates. Furthermore, candidates are assigned with properties that specify the node kind, tag and/or value. Candidate $c4$ is double circled to denote the result of the pattern.

3.3 From $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$

We map data from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ in order to represent a tree without edges. This mapping replaces edges between nodes with node annotations such that for each two nodes, their relative location in a tree with respect to each other is preserved. This approach follows the XA approach. Note that $\mathcal{U}_{row}$ is $\mathcal{U}$ that represents a table-structured data model.

Data mapping In order to express data in $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$, we take advantage of the work of Grust et al. [21]. A summary of their work is found in Section 2.3.2. They propose the XA approach; a mapping where each node is mapped to a set of properties such that the tree-structure is preserved. We apply the XA approach to $\mathcal{U}_{node}$ in order to substitute edges for property assignments. We consider the application of XA to an instance of $\mathcal{U}_{node}$ as mapping F from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ in Figure 3.1.

For illustration purposes, Figure 3.5 presents the result of F applied to the $\mathcal{U}_{node}$ instance in Figure 3.3. We observe a set of nodes annotated with a set of property assignments and a set of RVAs. Since the world set descriptors of objects are unaltered, it is easy to show that the same information is represented under F.

Query mapping Grust et al. [21] specify a set of range conditions that allow for the evaluation of tree-relationships on trees represented with the XA encoding. We give a summary of these range conditions in Section 2.3.2. We translate patterns specified for $\mathcal{U}_{node}$ to patterns specified for $\mathcal{U}_{row}$ with a substitution of XPath axes for the range conditions specified in Section 2.3.2. We refer to this translation as G.

For illustration purposes, Figure 3.6 shows the replacement operation performed by G. According to Section 2.3.2, the evaluation of the parent/child axis requires a level-attribute. In Section A.1 we provide a proof that allows for the evaluation of the parent/child axis as an SQL query that request for the closest ancestor node in a tree-structures with solely the properties pre-order rank and size.

3.4 From $\mathcal{U}_{row}$ to U-Rel

We map $\mathcal{U}_{row}$ data to U-Rel. Nodes in $\mathcal{U}_{row}$ are represented as rows in U-Rel such that node annotations are represented as row attributes. Both data structures make use of RVAs to describe the set of possible worlds of which their data entries are member.

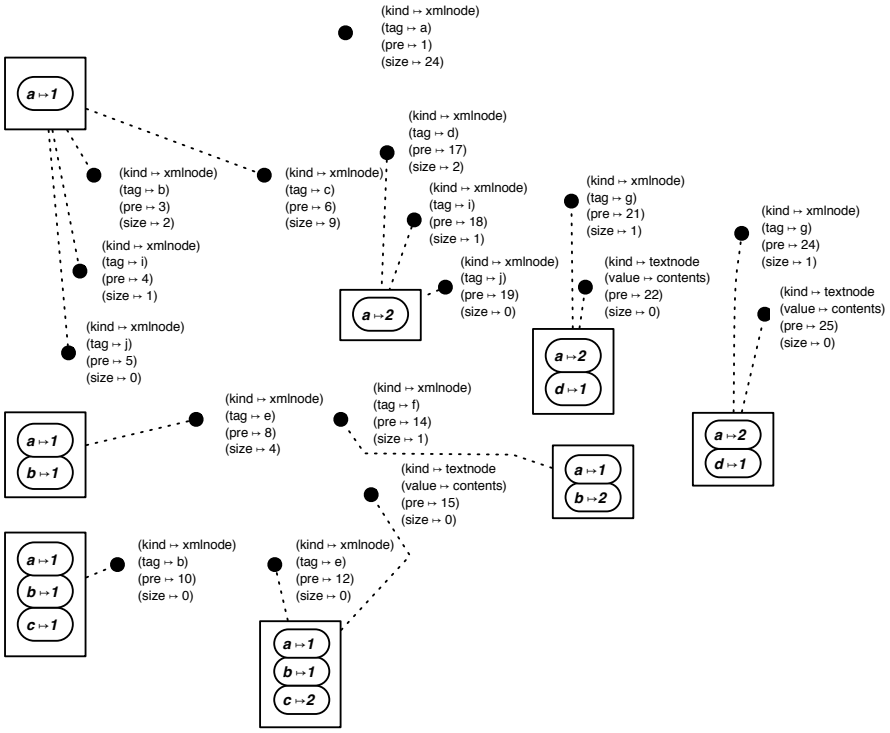


Figure 3.5: A $\mathcal{U}_{row}$ instance obtained by applying F to the $\mathcal{U}_{node}$ instance in Figure 3.3

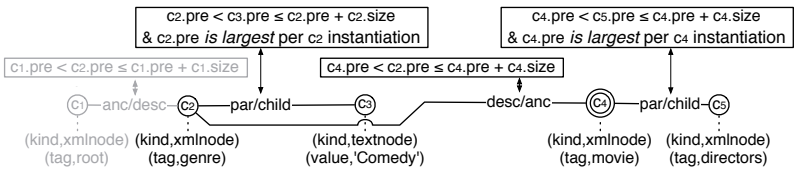


Figure 3.6: A substitution of range conditions for XPath axes as specified by query mapping G

Data mapping $\mathcal{U}_{row}$ contains a set of objects associated with properties and RVAs. We interpret objects as rows and the properties of objects as row-attributes. We copy the set of RVAs associated with an object to the row that represents that object. This mapping slightly changes the representation of the data but leaves the uncertainty distribution unaltered.

For illustration purposes, Figure 3.7 presents the U-Relation that represents Figure 3.5. RVAs are represented slightly different; random variable identifiers are stored in the three *var*-columns. Corresponding assignment values are stored in left adjacent *val*-columns. We refer to *val*-columns and *var*-columns as conditional columns. Row attributes of conditional columns are denoted as rectangles with rounded edges. Since the world set descriptors of objects are unaltered, it is easy to show that the same information is represented under rep_{row}.

pre	size	value	tag	kind	var_1	val_1	var_2	val_2	var_3	val_3
1	24	-	'a'	elem						
3	2	-	'b'	elem	a	1				
4	1	-	'i'	elem	a	1				
5	0	-	'j'	elem	a	1				
6	9	-	'c'	elem	a	1				
8	4	-	'e'	elem	a	1	b	1		
10	0	-	'b'	elem	a	1	b	1	c	1
12	0	-	'e'	elem	a	1	b	1	c	2
14	1	-	'f'	elem	a	1	b	2		
15	0	content	-	pcdata	a	1				
17	2	-	'd'	elem	a	2				
18	1	-	'i'	elem	a	2				
19	0	-	'j'	elem	a	2				
21	1	-	'g'	elem	a	2	d	1		
22	0	content	-	pcdata	a	2	d	1		
24	1	-	'h'	elem	a	2	d	2		
25	0	content	-	pcdata	a	2	d	2		

Figure 3.7: A U-Relation that represents the $\mathcal{U}_{row}$ instance in Figure 3.5

Query mapping We construct a mapping of patterns to SQL queries as follows. For each candidate, we add its corresponding U-Relation to the FROM-clause. Required properties and range conditions are specified as constraints on row-attributes in the WHERE-clause. The SELECT-clause holds those U-Relations that correspond with the set of candidates identified as the result of the pattern.

For illustration purposes, we present the SQL-query that corresponds with the pattern specified in Figure 3.6[1].

[1]A small note on performance: the relation between $c2$ to $c3$ and $c4$ to $c5$ is the parent/child relationship. This relation can be evaluated with a foreign key from the child to the parent as done by Grust et al. [20]. However, with the correct indices, our approach that searches for the local maximum has a similar performance. We show this with our benchmark in Chapter 12 where we compare precomputed chaining (PC).glue by possibility parent reference (PPR) –uses foreign key– with PC.glue by closest dependency (CD) –uses local maximum.

```
SELECT c5.*
FROM  urel c2,
      urel c3,
      urel c4,
      urel c5
WHERE c2.kind = "xmlnode"
AND   c2.tag = "genre"
AND   c2.pre < c3.pre
AND   c3.pre <= c2.pre + c2.size
AND   c2.pre = (
         SELECT max(c2_copy.pre)
         FROM  urel c2_copy
         WHERE c2_copy.pre < c3.pre
         AND   c3.pre <= c2_copy.pre + c2_copy.size
      )
AND   c3.kind = "textnode"
AND   c3.value = "Comedy"
AND   c4.pre < c2.pre
AND   c2.pre <= c4.pre + c4.size
AND   c4.kind = "xmlnode"
AND   c4.tag = "movie"
AND   c4.pre < c5.pre
AND   c5.pre <= c4.pre + c4.size
AND   c4.pre = (
         SELECT max(c4_copy.pre)
         FROM  urel c4_copy
         WHERE c4_copy.pre < c5.pre
         AND   c5.pre <= c4_copy.pre + c4_copy.size
      )
AND   c5.kind = "xmlnode"
AND   c5.tag = "directors"
```

3.5 Summary

In this chapter, we sketched our approach to obtain a specification of (f, g) where f is a P-XML into URDBMS data mapping and g is an XPath to SQL mapping. We take the approach to specify (f, g) as a sequential composition of four less complex mappings. Figure 3.1 gives an high level overview of this approach where each database mapping is represented as a double headed arrow. Since we defined a database mapping as a data mapping and a query mapping, we can unfold Figure 3.1 to Figure 3.8[2]. This unfolded figure has the shape of a cuboid constructed of four triangular prisms.

Our goal is to specify f and g such that the bottom side of Figure 3.8 commutes. All other sides denote mappings that we described in previous sections. In order to show that the front side and the back side commute, we take the approach to show for each data mapping f_i other than f that (1) the same set of possible worlds is represented, and (2) a query returns the similar equivalent result under the query mapping associated to f_i. If we specify f as the sequential composition of all f_i and g as the sequential composition of all corresponding query mappings, we obtain a correct P-XML into URDBMS database mapping.

[2] A slight difference, the intermediate data structure C-XML is not drawn.

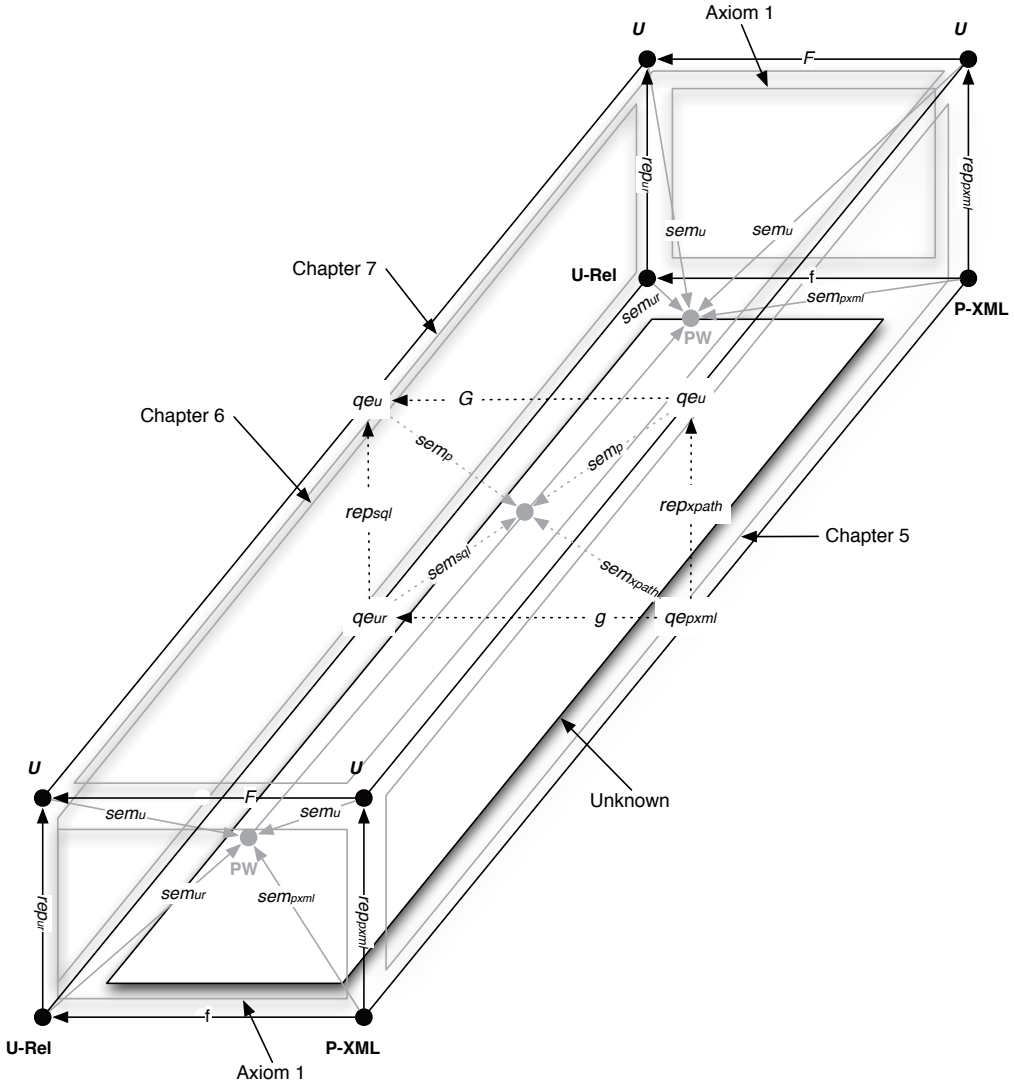


Figure 3.8: Figure 1.1 unfolded

Part II

Specification

Chapter 4

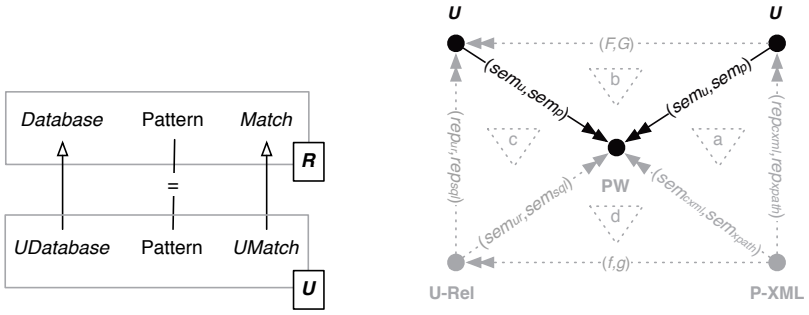
Abstraction of an Uncertain Data Model

In this chapter, we define an abstract formalism of an uncertain data model $\mathcal{U}$. First, we introduce a formalism $\mathcal{R}$ of a traditional data model. Second, we extend $\mathcal{R}$ with a similar uncertainty management mechanism as MayBMS in order to obtain $\mathcal{U}$. This extension is visualized in Figure 4.1a.

In the first part of this chapter, we introduce a formalism $\mathcal{R}$ of a traditional data model. We construct $\mathcal{R}$ as three concepts: (1) a generalization for data structures is captured in the concept of a *Database*, (2) a generalization for queries is captured in the concept of a *Pattern*, and (3) query evaluation with the two former mentioned concepts is captured with the concept of a *Match*. We use $\mathcal{R}$ to represent tree-structured data models and table-structured data models.

In the second part of this chapter, we introduce a formalism $\mathcal{U}$ of an uncertain data model. $\mathcal{U}$ extends $\mathcal{R}$ with the concept of a *U-Database* and the concept of a *U-Match*. These two new concepts extend $\mathcal{R}$ with a similar uncertainty management mechanism as MayBMS –which is a URDBMS. As a consequence, research contributions to the uncertainty management mechanism of MayBMS also apply to $\mathcal{U}$. We use this to show that $\mathcal{U}$ adheres to the possible worlds semantics. We use $\mathcal{U}$ to represent uncertain tree-structured data models and uncertain table-structured data models. We write $\mathcal{U}_{node}$ to refer to $\mathcal{U}$ that represents an uncertain tree-structured data model. Analogously, we write $\mathcal{U}_{row}$ to refer to $\mathcal{U}$ that represents an uncertain table-structured data model. The goal is to define a mapping from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ such that we express an uncertain tree-structured data model in an uncertain table-structured data model.

Figure 4.1b provides an overview of the contributions of this chapter related to the overall approach to design a correct P-XML into URDBMS mapping.



(a) $\mathcal{U}$ extends $\mathcal{R}$ (b) $\mathcal{U}$ in the bigger picture

Figure 4.1: Overview of the contributions of Chapter 4

4.1 Analogy with pattern matching

Pattern matching can be described as the act of checking a sequence of tokens for the presence of the constituents of some defined structure or pattern. If we consider tokens to be objects in a database and the sequence of tokens to be the structure encapsulated by database objects, we can interpret a database query as a pattern to which database objects are subjected. Each submission of a subset of objects to a pattern is referred to as a *combination*. In case the objects of

a combination match a pattern, we consider the combination to be an *instantiation* of that pattern. The *image of a pattern on that database* is defined as all instantiations of that pattern. The desired structure specified in a pattern and the structure of objects in a database are expressed in terms of *properties* of objects and *relations* between objects. Hence, more expressive structures can be expressed compared to solely the order of tokens in pattern matching. With the close analogy between a database query and a pattern, query evaluation can be interpreted as pattern matching: instead of searching for a sequence of tokens that match a certain pattern, we search all sets of objects in the contents of a database that match some structure specified in a pattern.

An illustration is found in Figure 4.2. Objects are denoted in the database with black circles that can be related to each other through some relation, in this case we use the relations r_1 and r_2. In this example, we have not taken properties into account. A query evaluated on a database is defined as a pattern of *candidates*, denoted with open circles on which required relations are specified. Double-circled candidates are defined to be the *result* of a pattern. A combination is defined as a one-to-one correspondence between candidates in a pattern and objects in a database. In case the objects of a combination have the same properties and relations as the candidates with which they are associated, the combination instantiates the pattern. In this example, we count four instantiations of the pattern. The image of a pattern on a database is described as the set of all instantiations of a pattern and is considered to be the result of the corresponding match. We illustrate a match as the original database where objects and relations that are not part of any instantiation are colored grey. It follows that the result of the pattern are the four double-circled objects.

4.2 An abstract formalism of a data model

In this section, we define the concepts *Database*, *Pattern* and *Match*. These three concepts together form $\mathcal{R}$, a formalism of a data model.

Assumptions. We postulate[1] the following sets of *objects*, *relation names*, *property names* and *values*:

$$[OBJ, REL, PROP, VAL]$$

For illustrative purposes, we postulate a few instances of these sets:

$$\begin{array}{l} o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8 : OBJ \\ colname : PROP;\ color, size : VAL \\ attrvalue : PROP \\ table : PROP;\ t1, t2 : VAL \\ row : PROP;\ r1, r2 : VAL \\ red, white, blue : VAL \\ small, medium, big : VAL \\ \hline \#\{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\} = 8 \\ \#\{red, white, blue\} = \\ \quad \#\{small, medium, big\} = 3 \\ \#\{color, size\} = 2 \end{array}$$

With the introduction of different universes, we assume they are disjoint. Therefore, a comparison between instances of different universes is not meaningful. We can check objects, properties and values on equality, however, we do not capture if the values *red* and *small* are different or not.

Objects can have properties with associated values which are presumed to be universal. In contrast, relations between elements are only relevant in the context of a database. We define the function *getProp* that captures the properties and associated values of each object:

[1] A *postulated* collection contains elements of which we only know how to check on equality.

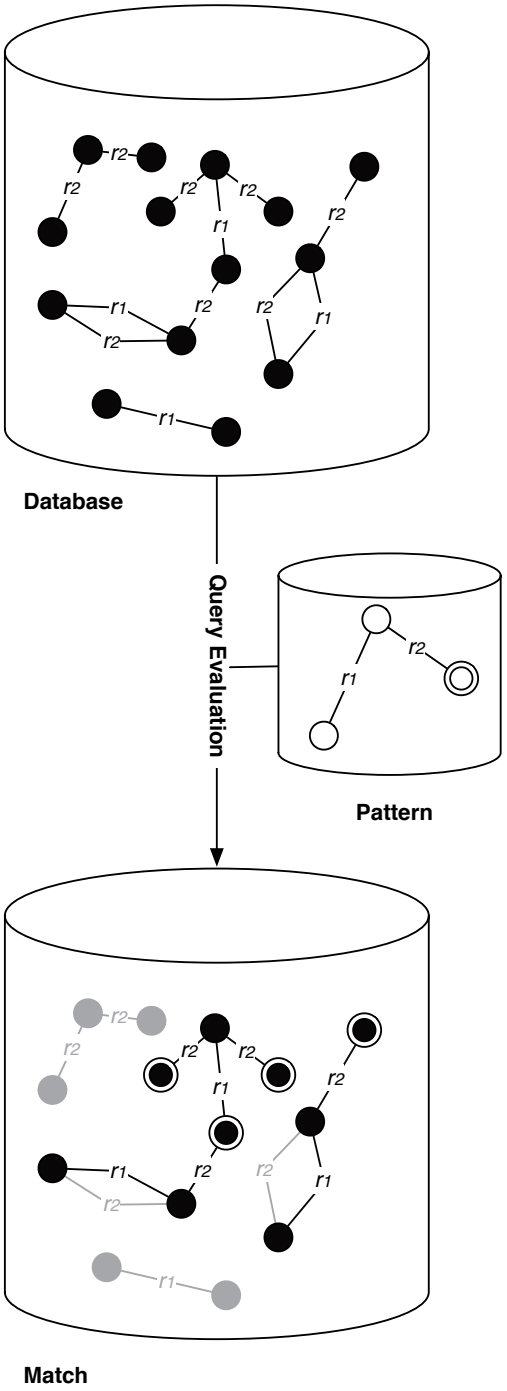


Figure 4.2: Query Evaluation $\sim$ Pattern Matching

$$
\begin{array}{|l}
getProp : OBJ \longrightarrow (PROP \longrightarrow VAL) \\
\hline
getProp = (OBJ \times \varnothing) \ \oplus \\
\quad \{o_1 \mapsto \{(table \mapsto t1), (row \mapsto r1), (colname \mapsto color), (attrvalue \mapsto red)\}, \\
\quad o_2 \mapsto \{(table \mapsto t1), (row \mapsto r1), (colname \mapsto size), (attrvalue \mapsto medium)\}, \\
\quad o_3 \mapsto \{(table \mapsto t1), (row \mapsto r2), (colname \mapsto color), (attrvalue \mapsto blue)\}, \\
\quad o_4 \mapsto \{(table \mapsto t1), (row \mapsto r2), (colname \mapsto size), (attrvalue \mapsto medium)\}, \\
\quad o_5 \mapsto \{(table \mapsto t2), (row \mapsto r1), (colname \mapsto color), (attrvalue \mapsto white)\}, \\
\quad o_6 \mapsto \{(table \mapsto t2), (row \mapsto r1), (colname \mapsto size), (attrvalue \mapsto big)\}, \\
\quad o_7 \mapsto \{(table \mapsto t2), (row \mapsto r2), (colname \mapsto color), (attrvalue \mapsto white)\}, \\
\quad o_8 \mapsto \{(table \mapsto t2), (row \mapsto r2), (colname \mapsto size), (attrvalue \mapsto big)\} \ \}
\end{array}
$$

The part $OBJ \times \varnothing$ specifies an empty collection for *every* object; the $\oplus$-operator "overwrites" this specification with some property assignments for the objects $o_1, \ldots, o_8$.
Notice that different objects can have identical properties. Examples are objects o_6, o_8. It can be proven that:

$$o_6 \neq o_8 \quad \wedge \quad getProp(o_6) = getProp(o_8)$$

We introduce the short hand notation $c.prop = getProp(c)(prop)$.

The concept of a Database. A *database over X* consists of a subset of X which we call *contents*, along with relations between elements. The set *relSet* captures these relations and assigns a relation name in *REL* to it. Relations between elements are in general undirected. However, in some scenarios, it is desirable to specify directed relations. In future chapter, we use the /-sign to specify a directed relation from left to right and vice versa. For example, the relation 'isParentOf/isChildOf' specified for objects o and o' should be read as: "o is parent of o'" and "o' is child of o". We use directed relations in future chapters.

$$
\begin{array}{|l}
\hline
Database[X] \\
contents : \mathbb{P}\, X \\
relSet : \mathbb{P}(X \times X \times REL) \\
\hline
relSet \subseteq \mathbb{P}(contents \leftrightarrow contents) \times REL \\
\hline
\end{array}
$$

As an example, we express the relational database in Figure 4.3 in $Database[X]$. A relational database stores table-structured data. A table structure consists of rows and columns of which the intersection of a row with a column is referred to as an *attribute*. We consider rows to be the objects in our database and attributes to be properties of the row they are associated with. Hence, we choose OBJ as database parameter X. Furthermore, a relational data structure implies the existence of *samerow* and *sametable* relations between attributes. These relations are considered known to a relational database:

$$
\begin{array}{|l}
db_0 : Database[OBJ] \\
\hline
db_0.contents = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\} \\
db_0.relSet = \\
\quad \{o, o' : db_0.contents \mid o, o' \in \ \wedge\, o.table = o'.table \bullet (o, o', sametable)\} \cup \\
\quad \{o, o' : db_0.contents \mid o, o' \in \ \wedge\, o.table = o'.table \ \wedge\ o.row = o'.row \\
\qquad \bullet (o, o', samerow)\}
\end{array}
$$

Note that the properties of these objects are already universally defined above.

The concept of a Pattern. A *pattern* specifies a number of "objects to be chosen" –referred to as *candidates*– with required mutual relations –referred to as *requiredRel*– and required properties –referred to as *requiredProp*. Furthermore, a pattern specifies a subset of candidates as "the answer"

Figure 4.3: Example of correspondence between row-attributes and objects

–referred to as *answer*. The evaluation of a pattern on a database – referred to as *match* – evaluates each combination of *contents* of a database in order to determine if a combination is an instantiation of a pattern.

We introduce a collection of candidates:

$$[CANDIDATE]$$

We reuse the schema of a database to define a *pattern* as a desired piece of a database. To this extent, we replace the terminology *contents* and *requiredRel* in the pattern schema with the terminology *candidates* and *requiredRel* (the notation for a replace-operation is: new/old) in order to use the concepts *Pattern* and *Database* simultaneously.

$$\begin{array}{|l} \textit{Pattern} \\ \hline Database[CANDIDATE][candidates/contents, requiredRel/relSet] \\ requiredProp : CANDIDATE \pfun \mathbb{P}(PROP \times VAL) \\ answer : \mathbb{P}\ CANDIDATE \\ \hline answer \subseteq candidates \\ \mathrm{dom}\ requiredProp = candidates \\ \hline \end{array}$$

The *answer* in this pattern schema can be interpreted as a special kind of projection applied to the desired piece of a database. The function *requiredProp* is introduced as a partial function (above the horizontal line). However, the reduction of the domain of this function (below the horizontal line) implies that *requiredProp* is a total function on *candidates*:

$$requiredProp \in candidates \fun \mathbb{P}(PROP \times VAL)$$

As an extension to our example database, we introduce a pattern-instance p_0:

$$\begin{array}{|l} c_1, c_2, c_3 : CANDIDATE \\ p_0 : Pattern \\ \hline p_0.candidates = \{c_1, c_2, c_3\} \\ p_0.requiredRel = \{(c_1, c_2, samerow), (c_2, c_3, sametable)\} \\ p_0.requiredProp = \{(c_1 \mapsto \{(colname, size), (attrvalue, big)\}), \\ \quad (c_3 \mapsto \{(colname, size)\})\} \\ p_0.answer = \{c_2, c_3\} \end{array}$$

To elaborate on this example, we specify a desired database structure as three candidates. In case a combination of database objects instantiates a *Pattern*, we do not return all objects that match, only the objects that correspond to candidates c_2 and c_3. The evaluation of this pattern on database db_0 is showed after we introduce the concept of a *Match*.

The concept of a Match. The evaluation of a pattern on a database –referred to as *Match*– is as follows: all possible combinations of objects in a database are evaluated in order to find all

combinations that instantiate a pattern. *If* for each combination, denoted with $f \in candidates \longrightarrow contents$, the *requiredRel* "after translation by f" is part of *relSet*, and the *requiredProp* "after translation by f" is part of *getProp*, *then* the combination instantiates the pattern that is evaluated. Its candidate-subset *answer* "after translation by f" is part of the final result. In any other case, the combination is rejected.

In order to formulate "after translation by f" in a compact fashion, we introduce two auxiliary functions *translP* and *translR*. Their definition is 'generic', denoted with a dual horizontal line, meaning that during a use of these functions, the argument types of X, Y, Z are automatically derived by a type-checker and do not have to be explicitly provided.

$$
\begin{array}{|l}
\hline
[X, Y, Z] \\
\hline
translP : (X \longrightarrow Y) \longrightarrow \mathbb{P}(X \times Z) \longrightarrow \mathbb{P}(Y \times Z) \\
translR : (X \longrightarrow Y) \longrightarrow \mathbb{P}(X \times X \times Z) \longrightarrow \mathbb{P}(Y \times Y \times Z) \\
\hline
translP\, f\, xzSet = \{x : X; z : Z \mid (x, z) \in xzSet \bullet (f\, x,\ z)\} \\
translR\, f\, xxzSet = \{x, x' : X; z : Z \mid (x, x', z) \in xxzSet \bullet (f\, x,\ f\, x',\ z)\} \\
\hline
\end{array}
$$

We apply both functions in the subsequent definition of *match* in which X is a collection with elements characterized as known/to be present (notice the parallel with the *has*-functions) versus Y, a collection with elements characterized to be required/desired (notice the parallel with the *required*-functions). Collection Z is used to evaluate a possible match between X and Y. Members of Z are the elements on which a match is evaluated; the comparison material so to speak. The *translP*-function uses Z to hold properties, while Z contains relations when the *translR*-function is applied. Finally, in case a subset of contents that forms a candidate possess the required properties and relations defined in a pattern, we speak of an *instantiation of the pattern.* Each database object that is involved in the instantiation of the pattern is said to *instantiate a candate.* We refer to the full set of these database objects as *candidate objects.* The candidate objects corresponding with the answer of a pattern are added to the *result* of the match.

$$
\begin{array}{|l}
\hline
Match \\
Database[OBJ] \\
Pattern \\
result : \mathbb{P}\ OBJ \\
\hline
o \in result \iff (\exists f : candidates \longrightarrow contents; a : answer \bullet \\
\qquad translP\, f\, requiredProp \subseteq getProp \land \\
\qquad translR\, f\, requiredRel \subseteq relSet \land \\
\qquad f\, a = o) \\
\hline
\end{array}
$$

It can be proven that the *result* of a match, and therefore the final answer to a pattern on a database, can be defined as follows:

$$
\begin{array}{l}
\forall\, Match \bullet \\
result = \bigcup \{f : candidates \longrightarrow contents \\
\qquad\qquad \mid translP\, f\, requiredProp \subseteq getProp \ \land\ translR\, f\, requiredRel \subseteq relSet \\
\qquad\qquad \bullet\, f(\!|answer|\!)\}
\end{array}
$$

The special braces $(\!|$ and $|\!)$ denote the image under f. Hence, $f(\!|answer|\!) = \{a : answer \bullet f(a)\}$.

To illustrate this, we provide the corresponding *match*-result of pattern p_0 on database db_0:

$$\begin{aligned}
&\forall\, Match \bullet \\
&\quad contents = db_0.contents \wedge \\
&\quad relSet = db_0.relSet \wedge \\
&\quad candidates = p_0.candidates \wedge \\
&\quad requiredProp = p_0.requiredProp \wedge \\
&\quad answer = p_0.answer \\
&\Longrightarrow \\
&\quad result = \{o_5, o_6,\ o_5, o_8,\ o_7, o_6,\ o_7, o_8\} = \{o_5, o_6, o_7, o_8\}
\end{aligned}$$

Proof: p_0 demands the existence of candidates c_1, c_2, c_3 with the requirement that "c_1 has a column name *size*" and "c_1 has *big* as attribute value". Objects o_6 and o_8 are the only two objects that satisfy these requirements. Hence, if any instantiation of the pattern should exist, one of these two objects is part of it. We say, o_6 and o_8 instantiate c_1. In addition, the relation "c_1 and c_2 are in the same row" specifies an object associated with c_1 to be in the same row with another object associated with c_2. If we consider o_6 to instantiate c_1, candidate c_2 can be instantiated with object o_5 since o_5 and o_6 share the same row. Analogously, o_8 shares a row with o_7. Hence, o_5 and o_7 instantiate c_2. Finally, candidate c_3 is in the relation "c_2 and c_3 share the same table". It has the property "c_3 has a column name *size*". If we consider the instantiation of c_2 with o_5, it follows that objects o_6 and o_8 instantiate c_3 since both objects satisfy the relation-requirement and the property-requirement of c_3. Likewise, we retrieve the objects o_6 and o_8 to instantiate c_3 if we consider the instantiation of c_3 with o_7. An instantiation of all three candidates result in an instantiation of the pattern. It follows that the following four combinations all instantiate pattern p_0: (1) o_6, o_5, o_6, (2) o_6, o_5, o_8, (3) o_8, o_7, o_6 and (4) o_8, o_7, o_8. The answer of p_0 is defined as the second and third element of the matching objects. Hence, we obtain the result $\{o_5, o_6, o_7, o_8\}$.

The concept of a property expression Members of *REL* are used to express relations of the form $X \times X \times REL$. For a database *db* and a pattern p, a pair of objects $\{o, o'\} \subseteq db.contents$ satisfy a required relation $(c, c', r) \in p.reqRel$ if $\exists f : p.candidates \longrightarrow db.contents$ such that $f(c) = o$, $f(c') = o'$ and $(f\ c, f\ c', r) \in db.relSet$.

We introduce property expressions as a special type of relation that defines a property comparison for a pair objects. For a property expression $pe \in REL$, a pair of objects $\{o, o'\}$ in $db.contents$ satisfy a required relation (c, c', pe) in $p.reqRel$ if $\exists f : p.candidates \longrightarrow db.contents$ such that $f(c) = o$, $f(c') = o'$ and $pe[c/f(c), c'/f(c')]$ holds. The slash is the notation for a replace-operation: *find*/*replace*.

As an example, pattern p has the following required property:

$$(c, c', c.pre < c'.pre) \in p.requiredRel$$

We identify '$c.pre < c'.pre$' as a member of *REL*. A combination $\{o, o'\}$ satisfies the propery expression if $o.pre < o'.pre$ holds. The expression $o.pre$ is short hand notation for $getProp(o)(pre)$, $o'.pre$ analogously.

Conclusion The concepts *Database*, *Pattern* and *Match* define $\mathcal{R}$. An interpretation for objects is required such that objects represent the database 'atoms" of the data model it represents. In Section 4.5, we extend $\mathcal{R}$ to cope with uncertainty.

4.3 The U-Relational model adheres to the possible worlds semantics

MayBMS extends PostgreSQL –an RDBMS– with an uncertainty management mechanism. This mechanism extends the data model and the query evaluation mechanism of PostgreSQL such that

MayBMS is a URDBMS for which query evaluation and data storage adhere to the possible world semantics [5]. In other words, data stored in the uncertain data model of MayBMS represent a set of possible worlds. We write U-Rel to refer to this model. The result of a traditional SQL query evaluated on data in U-Rel represents a set of possible answers, one provided by each of the represented possible worlds. Query evaluation and data storage with MayBMS are in more detail discussed in the work of Antova et al. [5]. We consider the double headed arrow from U-Rel to PW in Figure 4.1b to be proven.

U-Relations are uncertain relational tables for which each row is associated with a *WSD* which is a set of RVAs. An RVA is an uncertainty annotation of the form $(v \mapsto d)$ where v is the identifier of a *random variable* and d is a value assigned to v. We refer to values assigned to random variables as *assignment values*. A random variable represent represents a choice among mutually exclusive alternatives represented by the domain of values that can be assigned to that random variable. The interpretation of a random variable assignment is the designation of an alternative to be the correct one. The term "world set descriptor" originates from the fact that a set of RVAs describe a set of possible worlds. The WSD assigned to a row describes the set of possible worlds –or the *world set*– of that row.

For illustration purposes, we assume two rows r_1 and r_2. WSD $\{(x \mapsto 1)\}$ is assigned to r_1 and WSD $\{(x \mapsto 2)\}$ is assigned to r_2. As a consequence, r_1 is part of those possible worlds that assign value 1 to variable x, r_2 analogously. It follows that r_1 and r_2 cannot simultaneously be part of any possible world since the values 1 and 2 cannot be assigned to variable x at the same time.

U-Rel has the following properties [5]:

- *Expressiveness*: U-Relations are *complete* for finite sets of possible worlds. In other words, each finite set of possible worlds can be represented with a U-Relation.
- *Succinctness*: U-Relations support tuples to have multiple representations. Single possible representations of tuples are stored as single tuples in U-Relations. This is called vertical partitioning. Vertical partitioning and the adopting of tuples as unit of uncertainty allows for succinct data storage [2].
- *Purely relational*: U-Relations allow a rich class of queries that consist of operations in the positive relational algebra. The translation of queries to relational algebra is simple. As a consequence, complex queries are translated efficiently and correctly. Hence, a large subset of SQL constructs is able to deal with U-Relations. This allows the leveraging of RDBMS technology. In addition, since U-Relations are by their nature relations extended with world set description information stored in additional columns, U-Relations are purely relational.

4.4 Analogy with pattern matching continued

In Section 4.1, we introduce an analogy with pattern matching that shows many similarities with our abstract formalism $\mathcal{R}$ of a traditional data model. We extend this analogy with a notion for uncertainty to provide an analogy for the abstract formalism of an uncertain data model which we introduce in this following section.

Figure 4.4 illustrates two new concepts: the concept of a *U-Database* and the concept of a *U-Match*. The concept of a *U-Database* differs from the concept of a *Database* in the sense that objects are associated with WSDs. As we described in Section 4.3, WSDs describe the world-set of objects with which they are associated. World set descriptors are drawn as stacked rounded

[2] According to Antova et al. [5], U-Relations represent uncertainty at the attribute level which gives U-Relations an advantage for both query evaluation and succinct storage. This statement is not completely correct. In essence, if two tuples have a conflict on attribute level, these conflicting tuples are repaired with a WSD attached to both tuples. As a consequence, they are part of disjoint sets of possible worlds. In other words, tuples that have a conflict on attribute level are repaired on tuple level. Hence, MayBMS uses tuples as unit of uncertainty. The expressiveness of uncertainty between URDBMSs that use tuples as unit of uncertainty vs. URDBMSs that use attributes as unit of uncertainty do not differ. However, since attributes are more fine grained than tuples, the level of succinctness differs between these two URDBMS categories.

rectangles. Each rounded rectangle represents an RVA. These RVAs are used during the evaluation of a pattern to verify if the objects of a combination are part of multiple possible worlds. This verification is illustrated in the *U-Match* concept. The WSDs of the three leftmost objects are revealed. Arrows between RVAs denote a common use of a random variable. In case the union of all world-sets does not have conflicting RVAs, than there exist a possible world of which the objects of a combination are member. In this example, we observe conflicting RVAs since values 3 and 6 cannot be associated to random variable z at the same time. While the three left most objects comply with the specified pattern, they cannot occur simultaneously in a possible world. Hence, these objects do not instantiate the pattern.

In summary, instances of *U-Database* keep track of what objects reside in which possible worlds by means of a WSD that describes the world set of which an object is member. As a consequence, pattern-answers derived from objects in a *U-Database* reside in the intersection of those world sets. Objects that are part of a pattern-answer are assigned with a new WSD that describes this intersection of sets of possible worlds. Hence, *U-Matches* preserve a notion of *lineage*, also called *provenance information*: *"information that describes the origin and history of data in its life cycle"* [12]. In this definition, information that describes the origin and history of a *pattern answer* refers to the set of possible worlds of which all the candidate objects are part of that made the construction of that specific pattern-answer possible.

4.5 An abstract formalism of an uncertain data model

In this section, we introduce the concept of a *U-Database* that extends the concept of a *Database* – introduced in Section 4.2– with a similar uncertainty management system as the uncertainty management system of MayBMS. This concept captures the data storage for our formalism $\mathcal{U}$ of an uncertain data model. Query evaluation on *U-Database* is discussed in the following section with the concept of a *U-Match*. The concepts *U-Database*, *Pattern* and *U-Match* define $\mathcal{U}$.

Assumptions. We postulate the following sets of *random variables* and *values*:

$$[RVAR, RVAL]$$

For illustrative purposes, we postulate a few instances of these sets:

$$\begin{array}{|l} x, y, z : RVAR \\ v_1, v_2, v_3 : RVAL \\ \hline \#\{x, y, z\} = \\ \#\{v_1, v_2, v_3\} = 3 \end{array}$$

The concept of a U-Database :

$$\begin{array}{|l} \textit{U-Database}[X] \\ \textit{Database}[X] \\ rvaSet : \mathbb{P}(RVAR \times RVAL) \\ getWSD : X \pfun \mathbb{P}(RVAR \times RVAL) \\ \hline \bigcup(getWSD(\!|contents|\!)) \subseteq rvaSet \\ \forall\, o : contents; rva, rva' : rvaSet \mid \{rva, rva'\} \subseteq getWSD\ o \bullet \\ \qquad first\ rva = first\ rva' \implies second\ rva = second\ rva' \end{array}$$

We state that a U-Relation uses the schema of a database and extends it with an *rvaSet* that stores random variable assignments. The function *getWSD* associates each object in the database with a WSD; a set of RVAs for which two conditions have to hold. The first condition states that all RVAs associated to database objects have to be known by the database. The second condition

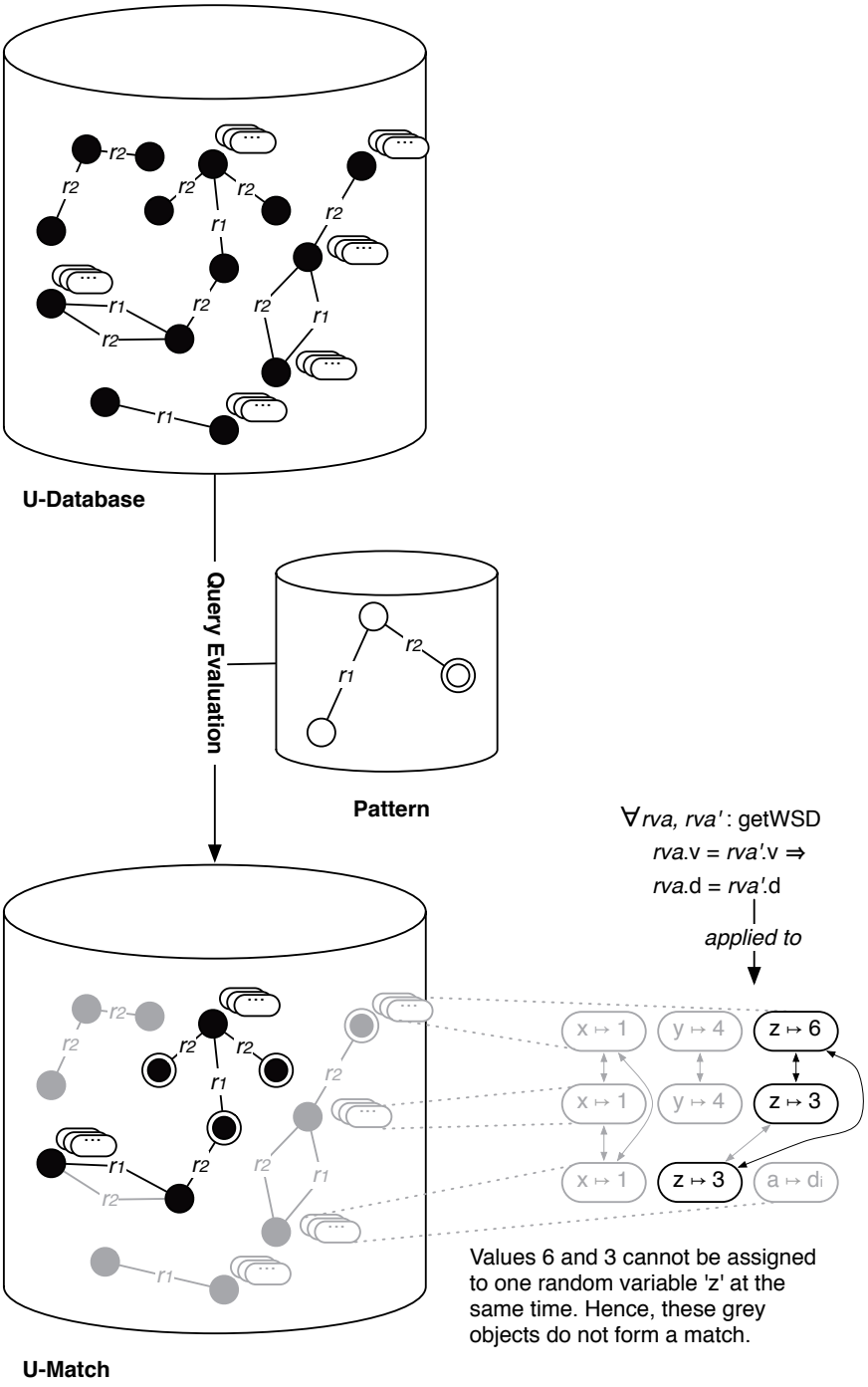


Figure 4.4: Evaluation of a Pattern on Uncertain Data

states that each instance in the database cannot be associated with conflicting RVAs. If it would have conflicting RVAs, by definition, no possible world can be described with conflicting RVAs. The second condition can also be written as:

$$\forall\, o : contents \bullet getWSD\ o \in RVAR \longrightarrow RVAL$$

This condition states that each WSD has to be an element of a mapping-function from *RVAR* to *RVAL*. We introduce this third condition to get acquainted with the observations that a set of RVAs form a function. We use a similar condition to define evaluation of patterns on U-Relations later on.

The definition of schema *U-Database* does not fully comply with MayBMS. MayBMS extends each RVA with a probability that captures the likelihood of that RVA to be correct. For simplicity, we ignore probabilities. However, they can easily be integrated in the model.

We illustrate U-Relations with an example. We consider the same relations and objects of db_0 now stored in U-Relation udb_0. The main difference is a set of random variable assignments:

$$
\begin{array}{|l}
udb_0 : U\text{-}Database[OBJ] \\
\hline
udb_0.contents = \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8\} \\
udb_0.relSet = \\
\quad \{o, o' : db_0.contents \mid o, o' \in \ \wedge\, o.table = o'.table \bullet (o, o', sametable)\} \cup \\
\quad \{o, o' : db_0.contents \mid o, o' \in \ \wedge\, o.table = o'.table \ \wedge\ o.row = o'.row \\
\qquad \bullet\, (o, o', samerow)\} \\
udb_0.rvaSet = \\
\quad \{(x \mapsto v_1), (x \mapsto v_2), (x \mapsto v_3), \\
\quad (y \mapsto v_1), (y \mapsto v_2)\} \\
udb_0.getWSD = (OBJ \times \varnothing)\ \oplus \\
\quad \{\ o_5 \mapsto \{(x \mapsto v_1)\}, \\
\quad o_6 \mapsto \{(x \mapsto v_1)\}, \\
\quad o_7 \mapsto \{(x \mapsto v_1), (y \mapsto v_2)\}, \\
\quad o_8 \mapsto \{(y \mapsto v_2)\}\ \}
\end{array}
$$

We use this example U-Relation to illustrate evaluation of patterns after we introduced the concept *U-Match*, the uncertain variant of the match concept.

4.6 Query evaluation on uncertain data

As stated in Section 2.4.1, the semantics of a traditional query Q executed on a set of possible worlds evaluates to a set of possible answers, each provided by one of the possible worlds. MayBMS uses another more efficient approach to evaluate queries. This approach is inspired by Imielinski et al. [30]. For complete representation systems like U-Relations, it is proven that a query $\hat{Q}$ evaluated on a complete representation system produces the same result as query Q evaluated on the set of possible worlds represented by that complete representation system.

The query translation of Q to its counterpart $\hat{Q}$ is executed internally as part of the query evaluation mechanism. As a consequence, the original query language is supported. The translation of Q to its counterpart $\hat{Q}$ is illustrated in Figure 4.5.

The concept of a U-Match We apply the Imielinski translation to the concept of a *Match* which we call *U-Match*. As a consequence, evaluation of patterns on the concept of a *U-Database* is changed. However, the concept of a *Pattern* is not. The schema of a *U-Match* captures evaluation of patterns on U-Relations such that we adhere to the possible worlds semantics:

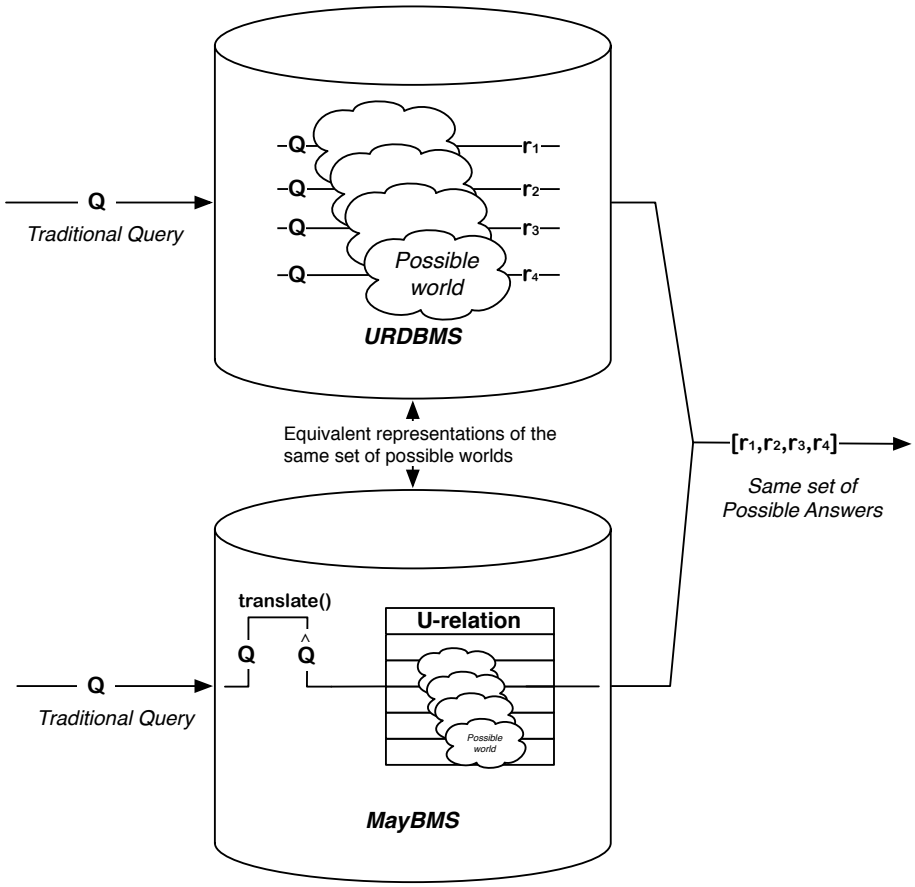


Figure 4.5: Imielinski translation in MayBMS

U-Match

$$
\begin{array}{l}
U\text{-}Database[OBJ] \\
Pattern \\
result : \mathbb{P}\ OBJ \\
WSDassignment : \mathbb{P}(OBJ \times (\mathbb{P}(RVAR \times RVAL)))
\end{array}
$$

$$
\begin{array}{l}
(o, g) \in WSDassignment \\
\iff \\
(\exists f : candidates \longrightarrow contents;\ a : answer \bullet \\
\quad translP\ f\ requiredProp \subseteq getProp \land \\
\quad translR\ f\ requiredRel \subseteq relSet \land \\
\quad f\ a = o \land \\
\quad g = \bigcup(getWSD\ (\!|\ f\ (\!|\ candidates |\!)\ |\!)) \in (RVAR \longrightarrow RVAL)) \\
result = \mathrm{dom}\ WSDassignment
\end{array}
$$

The *U-Match* concept defines the evaluation of queries on U-Relations. The result of a match is extended with a *WSDassignment*, that adds a notion of uncertainty to each entry in the result. The need for this function follows from the fact that a result is derived from candidate objects which all have to be part of a possible world in order for the pattern-answer to exist. Hence, these candidate objects are prohibited to have any conflicting RVAs. In the *U-Match* schema, the new WSD g of some pattern-answer o is defined as the union of all WSDs that are associated to the candidate objects of which o originates. We define the value that is assigned to g to be an element of the function $RVAR \longrightarrow RVAL$. This short notation ensures that g cannot contain conflicting RVAs. Hence, o resides in those possible worlds described by g. Since g does not contain conflicting RVAs, the new world set of o is not empty.

We finish this section with the evaluation of the previously introduced pattern p_0 on udb_0:

$$
\begin{array}{l}
\forall\ U\text{-}Match \bullet \\
\quad contents = udb_0.contents \land \\
\quad rvaSet = udb_0.rvaSet \land \\
\quad getWSD = udb_0.getWSD \land \\
\quad relSet = udb_0.relSet \land \\
\quad candidates = p_0.candidates \land \\
\quad requiredProp = p_0.requiredProp \land \\
\quad answer = p_0.answer \\
\Longrightarrow \\
\quad result = \{o_5, o_6,\ o_5, o_8,\ o_6, o_8\} = \{o_5, o_6, o_8\} \land \\
\quad WSDassignment = \{(o_5, \{(x \mapsto v_1)\}), (o_6, \{(x \mapsto v_1)\}), \\
\qquad (o_5, \{(x \mapsto v_1), (y \mapsto v_1)\}), (o_8, \{(x \mapsto v_1), (y \mapsto v_1)\}), \\
\qquad (o_6, \{(x \mapsto v_1), (y \mapsto v_1)\}), (o_8, \{(x \mapsto v_1), (y \mapsto v_1)\})\}
\end{array}
$$

Proof: if we ignore WSDs for a moment, we have the same scenario as p_0 evaluated on db_0. We obtain four combinations that can instantiate pattern p_0: (1) o_6, o_5, o_6, (2) o_6, o_5, o_8, (3) o_8, o_7, o_6, and (4) o_8, o_6, o_8. The proof of this is found in Section 4.2. However, since we ignore WSDs, it is possible that the WSD of one of these pattern instantiations describes an empty world set. This means that the objects of which a possible pattern instantiation is derived cannot be part of any possible world simultaneously. In order to validate WSDs to be non-empty, a WSD may not contain conflicting RVAs. Hence, we verify the WSDs of each of the four possible pattern instantiations to have no conflicting RVAs. If the verification is successful, the possible instantiation is a real instantiation. Otherwise, it is not an instantiation.

For the first possible instantiation, we obtain WSDs $\{(x \mapsto v_1)\}$, $\{(x \mapsto v_1)\}$ and $\{(x \mapsto v_1)\}$ for combination o_6, o_5, o_6. The union of these WSDs does not contain conflicting RVAs. For the second possible instantiation, we obtain WSDs $\{(x \mapsto v_1)\}$, $\{(x \mapsto v_1)\}$ and $\{(y \mapsto v_1)\}$ for combination o_6, o_5, o_8. Again, the union of these WSDs does not contain conflicting RVAs. For the third possible instantiation, we obtain WSDs $\{(y \mapsto v_1)\}$, $\{(x \mapsto v_1), (y \mapsto v_2)\}$ and $\{(z \mapsto v_1)\}$

for combination o_8, o_7, o_6. In the union of these WSDs, the RVAs $(y \mapsto v_1)$ and $(y \mapsto v_2)$ conflict. Hence, the third possible instantiation is not a real instantiation. For fourth and final possible instantiation, we obtain WSDs $\{(y \mapsto v_1)\}$, $\{(x \mapsto v_1)\}$, $\{(y \mapsto v_1)\}$. The union of these WSDs does not contain a conflicting RVAs.

In summary, of the four possible possible instantiations, actually three combinations instantiate the pattern. These three instantiations provide the following *U-Match* result: the first instantiation results in $(o_5, \{(x \mapsto v_1)\}), (o_6, \{(x \mapsto v_1)\})$; the second instantiation results in $(o_5, \{(x \mapsto v_1), (y \mapsto v_1)\}), (o_8, \{(x \mapsto v_1), (y \mapsto v_1)\})$; the fourth instantiation results in $(o_6, \{(x \mapsto v_1), (y \mapsto v_1)\}), (o_8, \{(x \mapsto v_1), (y \mapsto v_1)\})$. □

Observe that we have two different representations of objects o_5 and o_6. In case an object has different representations, its representations originate from different possible worlds. This is also the case for objects o_5 and o_6: one representation of o_5 originates from WSD $\{(x \mapsto v_1), (y \mapsto v_1)\}$ and its other representation orginates from WSD $\{(x \mapsto v_1)\}$

The concepts *U-Database*, *Pattern* and *U-Match* define our formalism of an uncertain data model. We abstract from any data structure such that various data structures can be expressed in one formalism. For future use, we refer to our formalism of an uncertain data model as $\mathcal{U}$. Parameter X refers to the "atoms" of a data structure and provides an interpretation for objects in *U-Database*.

4.7 $\mathcal{U}$ in the big picture

Motivation to design $\mathcal{U}$ In order to design a correct database mapping from P-XML to U-Rel, we use $\mathcal{U}$ as follows:

- We represent P-XML in $\mathcal{U}$. We write $\mathcal{U}_{node}$ to denote $\mathcal{U}$ that represents a tree-structure.
- We represent U-Rel in $\mathcal{U}$. We write $\mathcal{U}_{row}$ to denote $\mathcal{U}$ that represents a table-structure.

Our goal is to show that $\mathcal{U}_{node}$ and $\mathcal{U}_{row}$ represent the same set of possible worlds under our yet to define $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ mapping. Consequently, a mapping from P-XML to U-Rel based on a correct $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ mapping will also represent the same set of possible worlds.

A database instance of $\mathcal{U}$ represents a set of possible worlds The design of $\mathcal{U}$ is based on the work of Antova et al. [5] as described in Section 4.6. They show that a U-Relation represents a set of possible worlds and that query evaluation on U-Relations conforms to the possible worlds semantics. We give a description between the possible worlds model and $\mathcal{U}$.

We postulate the universe of possible worlds:

$$WORLD$$

A database instance $ud : U\text{-}Database$ represents a set of possible worlds:

$$sem : ud \longrightarrow \mathbb{P}\ WORLD$$

A possible world is characterized by a set of choices. In the schema of *U-Database*, choices are represented as RVAs:

$$getWorld : \mathbb{P}\ RVAR \times RVAL \longrightarrow WORLD$$

The schema of *U-Database* defines an instance to contain objects that are associated with a WSD. A WSD describes the set of possible worlds of which an object is member. We refer to this set of possible worlds as the world set of that object. A WSD is constructed as a set of RVAs. An RVA represents a choice that is made in some possible worlds represented by a database. Hence, a WSD represents a set of choices that are made in some possible worlds represented by a database.

$$getWorldSet : \mathbb{P}\ RVAR \times RVAL \longrightarrow \mathbb{P}\ WORLD$$

Such that:

$$\forall\, wID : \operatorname{dom} getWorld, wsID : \operatorname{dom} getWorldSet \bullet \\ wsID \subseteq wID \implies getWorld\ wID \in wsID\ getWorldSet$$

Represent the same set of possible worlds The schema of *U-Database* provides the link to represent a set of possible worlds as objects associated with a WSD. We would like a theorem with which we can show that two database instances of $\mathcal{U}$ represent the same set of possible worlds. Such theorem provides the means to test if the same set of possible worlds is represented under a mapping from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$.

We use the following theorem to show that two database instances represent the same set of possible worlds:

Axiom 1. *For each* $db, db' \in$ *U-Database:*

$$sem(db) = sem(db') \\ \iff \\ \exists f : db.contents \rightarrowtail\!\!\!\!\!\!\rightarrow db'.contents \bullet \\ \forall\, o \in db.contents \bullet getWorldSet(getWSD(o)) = getWorldSet(getWSD(f(o)))$$

Database instances db and db' represent the same set of possible worlds if (1) they represent similar database objects and (2) the world set of each database object o is the same as the world set of its counterpart $f(o)$. *We have the obligation to show that this axiom holds*

Final remark Axiom 1 provides a test with which two database instances of $\mathcal{U}$ can be validated to represent the same set of possible worlds. We use this theorem to validate if the same set of possible worlds is represented under our design of a $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ data mapping.

4.8 Summary

In this chapter, we define an abstract formalism of an uncertain data model to which we refer as $\mathcal{U}$. This data model is constructed as the concepts *U-Database*, *Pattern* and *U-Match* such capture data structure, query language and the semantics for query evaluation for $\mathcal{U}$. The design of $\mathcal{U}$ uses a similar uncertainty management mechanism as MayBMS. As such, $\mathcal{U}$ inherits compliance with the possible worlds semantics.

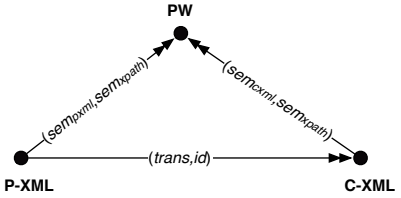
The design of the data structure of $\mathcal{U}$ gave raise to define Axiom 1. This theorem defines a test that verifies if two database instances of $\mathcal{U}$ represent the same set of possible worlds. We use this axiom in future chapters to define a data mapping from $\mathcal{U}_{node}$ –$\mathcal{U}$ that represents a tree-structured data model– to $\mathcal{U}_{row}$ –$\mathcal{U}$ that represents a table-structured data model.

Chapter 5

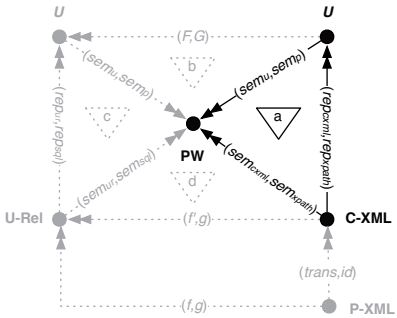
Probabilistic XML expressed in an Abstract Formalism

This chapter presents our advancing understanding to specify a mapping from P-XML to $\mathcal{U}$. Our only intention is to present our ideas of such mapping. We do not have the ambition to define a P-XML to $\mathcal{U}$ in detail.

We construct a mapping from P-XML to $\mathcal{U}$ in two parts. First, we specify a mapping from P-XML to C-XML –another data model. Second, we specify a mapping from C-XML to $\mathcal{U}$. This approach is illustrated in Figures 5.1a and 5.1b. First, database mapping $(trans, id)$ maps P-XML to C-XML. Second, database mapping (rep_{pxml}, g_{xpath}) maps C-XML to $\mathcal{U}$. A database mapping from P-XML to $\mathcal{U}$ is constructed as $(rep_{pxml} \circ trans, g_{xpath} \circ id)$.



(a) P-XML to C-XML mapping that adheres to the possible worlds semantics



(b) Extension of the initial proof obligation with the C-XML model

Figure 5.1: Overview of the contributions of Chapter 5

The C-XML data model has a strong resemblance with the P-XML data model. First, XML content is in both uncertain data models enriched with an uncertainty distribution such that their document instances represent a set of possible worlds. Document instances of P-XML are referred to as p-documents. Likewise, document instances of C-XML as *c-documents*. Second, both data models adopt the same query language as the XML data model. However, the way how query evaluation deals with two different uncertainty distributions differs.

The main difference between C-XML and P-XML is that the former defines an uncertainty distribution with CPAs –an XML kind of RVAs– assigned to nodes in a tree while the latter defines an uncertainty distribution with distributional node that reside in a tree. In order to bridge the gap between these two data models, we map distributional nodes to CPAs such that for each ordinary node, the same world set is described.

The main advantage of C-XML is that a set of possible worlds is represented in a similar fashion as in $\mathcal{U}$. Hence, a mapping between the two is more easily accomplished. We use C-XML to bridge the gap between P-XML and $\mathcal{U}$.

5.1 Probabilistic XML data structure

Definition of the P-XML data structure Schema P-XML defines the data structure of P-XML. Its instances are called p-document. This schema extends schema *ABS-XML* –introduced in Section 2.2.1:

PXML

$$
\begin{array}{l}
\textit{ABS-XML} \\
\textit{distrnodes}, \textit{possnodes}, \textit{probnodes}, \textit{allnodes} : \mathbb{P}\ \textit{NODE} \\
\textit{possChoices} : \textit{NODE} \leftrightarrow \textit{NODE} \\
\triangledown : \textit{NODE} \nrightarrow \textit{NODE} \times \textit{NODE} \\
\hline
\langle \textit{xmlnodes}, \textit{possnodes}, \textit{probnodes}, \textit{textnodes} \rangle \ \mathsf{partition}\ \textit{allnodes} \\
\textit{distrnodes} = \bigcup \{ \textit{possnodes}, \textit{probnodes} \} \\
\textit{rootnode} \in \bigcup \{ \textit{xmlnodes}, \textit{probnodes} \} \\
\mathrm{dom}\ \textit{edge} = \textit{allnodes} \setminus \{ \textit{rootnode} \} \\
\textit{possnodes} \lhd \textit{edge} = \textit{edge} \rhd \textit{probnodes} \\
\textit{possChoices} = (\textit{possnodes} \lhd \textit{edge})^{\sim} \\
\mathrm{dom} \triangledown = \textit{possnodes} \\
\mathrm{ran} \triangledown = \textit{possChoices} \\
\forall\, d : \textit{possnodes} \bullet \triangledown d = (\textit{edge}\ d \mapsto d)
\end{array}
$$

The extension of the XML data structure to the P-XML data structure is briefly discussed in Section 2.5.1. This extension is based on two new node kinds: possibility nodes *possnodes* and probability nodes *probnodes*. Possibility nodes and probability nodes are referred to as *distributional nodes*. Possibility nodes have a probability node as parent and the child axis of each probability node solely contains possibility nodes: $\textit{possnodes} \lhd \textit{edge} = \textit{probnodes} \rhd \textit{edge}$ where symbol $\lhd$ denotes a domain restriction and symbol $\rhd$ denotes a range restriction. The set of all possible alternative selections is captured with *possChoices* which is defined as a range restriction on the child axis. Each entry $(\textit{prob} \mapsto \textit{poss}) \in \textit{possChoices}$ represents a possible alternative selection for a choice point *prob* and alternative *poss* such that $\triangledown\, \textit{poss} = (\textit{prob} \mapsto \textit{poss})$.

Abstract definition of possible document A possible document is retrieved from a p-document with a subset of *possChoices* such that for each $\textit{prob} \in \textit{probnodes}$, one alternative selection is made. An alternative selection $(\textit{prob} \mapsto \textit{poss})$ should be interpreted as choice point *prob* to select *poss* as its one and only child. We capture these selections with the function *worldID*. Unselected alternatives are discarded. As a consequence, descendants of unselected alternatives are discarded as well.

Schema *AbsPD* extends the schema of P-XML. Its instances are called possible documents. The relation between *discardedAllnodes* and *worldID* is yet undefined. We define this relation in two schemes that define possible documents for P-XML and C-XML as an extension of *AbsPD*.

AbsPD

$$
\begin{array}{l}
\textit{P-XML} \\
\textit{worldID} : \textit{NODE} \nrightarrow \textit{NODE} \\
\textit{discardedAllnodes} : \mathbb{P}\ \textit{NODE} \\
\textit{xmlnodes}', \textit{distrnodes}', \textit{possnodes}', \textit{probnodes}', \textit{textnodes}', \textit{allnodes}' : \mathbb{P}\ \textit{NODE} \\
\hline
\mathrm{dom}\ \textit{worldID} = \textit{probnodes} \\
\textit{allnodes}' = \textit{allnodes} \setminus \textit{discardedAllnodes} \\
\textit{xmlnodes}' = \textit{xmlnodes} \setminus \textit{discardedAllnodes} \\
\textit{distrnodes}' = \textit{distrnodes} \setminus \textit{discardedAllnodes} \\
\textit{possnodes}' = \textit{possnodes} \setminus \textit{discardedAllnodes} \\
\textit{probnodes}' = \textit{probnodes} \setminus \textit{discardedAllnodes} \\
\textit{textnodes}' = \textit{textnodes} \setminus \textit{discardedAllnodes}
\end{array}
$$

In this schema, *discardedAllnodes* denotes the set of nodes that is discarded as a result of the choices made in *worldID*. Primed sets capture how sets in *P-XML* are affected by *worldID*.

Note that our definition of a possible document is slightly different than other work on P-XML. We consider a possible document as a p-document with a function *worldID* that selects for each

choice point one alternative. As a consequence, some possible documents may correspond with the same random document.

Definition of possible documents for P-XML We extend *AbsPD* in order to obtain schema $PD_{P\text{-}XML}$ that defines possible documents for P-XML:

$PD_{P\text{-}XML}$

AbsPD

$$discardedAllnodes = \bigcup\{prob, poss : NODE \mid (prob \mapsto poss) \in possChoices \\ \wedge\ worldID\ prob \neq poss \bullet poss/\texttt{descendant-or-self}\}$$

The definition of *discardedAllnodes* originates from the following. The set *P-XML.possChoices* captures a set of possible alternative selections. A subset defines the actual alternative selections and is captured by the function $PD_{P\text{-}XML}.worldID$. Since alternative selections are mutual exclusive, if a possible alternative selection in *possChoices* is unselected by $PD_{P\text{-}XML}.worldID$, the associated alternative and its descendants are discarded.

A possible document identifier In order to obtain a possible document from a p-document, one has to select one alternative for each choice point in a p-document and discard all unselected alternatives. We refer to the set of choices with which a possible document is obtained as the *world identifier* of that possible document. It follows from the definition of a possible document that, given a p-document, a world identifier points to exactly one possible document. Vice versa, all combinations of alternative selections that form a world identifier point to the set of possible documents represented by a p-document.

For illustration purposes, our running example in Figure 2.8 has 4 choice points which all have 2 alternatives. A total of 2^4 unique combinations can be made with which 2^4 possible documents are encoded. Note that some possible documents can correspond with the same random document (the XML-document that corresponds with a possible document).

A set of choices describes a set of possible documents A set of possible documents is identified with a set of choices that select one alternative for a subset of choice points in a p-document describe. We refer to such set as a world set descriptor (WSD). More specifically, given a p-document, a possible document *pd* is member of the set of possible documents described by a WSD if the choices specified in WSD form a subset of the choices specified in the world identifier of *pd*.

For illustration purposes, let $s = \{(v_1 \mapsto d_{1,2}), (v_2 \mapsto d_{2,1})\}$ be a WSD in Figure 2.8. Set s specifies two choice points to select one alternative. For the two remaining choice points, no alternative selections has been made. A total of 2^2 possible documents have made the choices specified in s, since 2^2 combinations can be made with the two remaining choice points.

5.2 P-XML adheres to the possible worlds semantics

Van Keulen et al. [51] show that document instances of their P-XML data model represent a set of possible worlds as a set of possible documents. Schema P-XML defines the P-XML data structure. This schema is found in the previous section. Additionally, we specify schema $PD_{P\text{-}XML}$ that defines the concept of a possible document. The combination of both schemes capture the relationship between P-XML and the possible worlds model.

Van Keulen et al. [51] describe the semantics of query evaluation on their data model. The result of a query evaluated on a set of possible worlds is a set of possible answers, each provided by one of the possible worlds separately. In order to evaluate a query on a p-document, one approach is to extract each possible world represented as a possible document, evaluate the query on each possible document, and merge the result. This approach adheres to the possible worlds semantics. We consider the double headed arrow from P-XML to PW in Figure 4.1b to be proven.

5.3 Viability

In this section, we introduce the concepts *viability*, *viable dependent* and *aliveness*. We use aliveness in future sections to show that two sets of possible worlds are similar.

The concept of viability. We say that an element $x{:}X$ is *viable* in a set of possible worlds iff there exists a possible world of which that element is a member.

In P-XML, a set of possible worlds is represented as a p-document such that each possible world corresponds with a possible document. As we explained in the previous section, the schema of a possible document extends the schema of P-XML with a function *worldID* that defines for each choice point one alternative to be selected. As a result, descendants of unselected alternatives are discarded. The set of choices with which a possible document is extracted from a p-document is unique for all possible documents represented by a p-document. A possible document is derived from a p-document by selecting alternatives for choice points until one alternative is selected for each choice points. An alternative selection applied to a p-document results in a new p-document that represents a smaller set of worlds until only one possible worlds remains.

If we consider a p-document for which no alternative selections are made, each node is viable. Proof sketch: if no alternative selections are made, no alternative is discarded. Hence, all possibility nodes are viable. For any node n, a possible document can be constructed of which n is part of. The construction of such document selects all alternatives that lie on path $\uparrow_n$. Such path exists, since all possibility nodes are viable. If path $\uparrow_n$ can be constructed, nodes that lie on path $\uparrow_n$ cannot be discarded and have to be viable, including n. □

The concept of viably dependent. If it holds that for each possible world of which an element $x{:}X$ is part, an $y{:}Y$ is also part of that possible world, we define x to be *viably dependent* on y and refer to y as a *viability dependency* for x. This is denoted as $x \parallel y$.

For P-XML, if a path $\uparrow_n$ can be constructed in a p-document, n is viable. Alternative selections that lie on $\uparrow_n$ are the viable dependencies of n. For illustration purposes, we represent $\uparrow_n$ in Figure 5.2 for some node n. In this example, n is part of those possible worlds in which d_i, d_j, d_k and d_l are selected. These alternative selections are the viable dependencies of n.

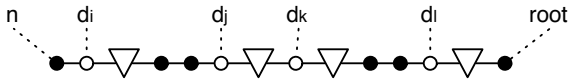


Figure 5.2: The path of a node n to the root of a p-document

The concept of aliveness. We define an element $x{:}X$ to be *alive* for a set of viability dependencies $ys : \mathbb{P}\ Y$ if in each possible world of which all elements of ys are member, x is also member. We write:

$$
\begin{array}{|l}
[X, Y] \\
\hline
(\_ \hat{\parallel} \_) : X \longrightarrow \mathbb{P}\ Y \\
\hline
\forall\, x : X \bullet x \hat{\parallel} ys \iff ys =\!\parallel (\!|\{x\}|\!)
\end{array}
$$

If we return to our example in Figure 5.2, n is part of those possible documents in which path $\uparrow_n$ is present. Path $\uparrow_n$ is part of those possible documents for which all alternatives that are part of $\uparrow_n$ are selected. It follows that n is alive for all alternative selections on $\uparrow_n$. In general, for *P-XML* and node $n \in$ *P-XML.ordnodes* holds:

$$n \hat{\parallel}_{P\text{-}XML} \{(prob \mapsto poss) \in possChoices \mid prob \in \uparrow_n \wedge poss \in \uparrow_n\}$$

The concept of a skeleton path We define skeleton path of node n –denoted as $\Uparrow_n$– as the path constructed of all distributional nodes that lie on $\uparrow_n$.

Theorem 1. *The skeleton path of a node n is the set of all viability dependencies for n. Let n be a node in* P-XML*. The following holds:*

$$n \hat{\parallel}_{PXML} (\Uparrow_n)$$

For illustration purposes, Figure 5.3 presents the skeleton path extracted from Figure 5.2. We observe four possible choices or alternative selections denoted as edges from possibility nodes to probability nodes.

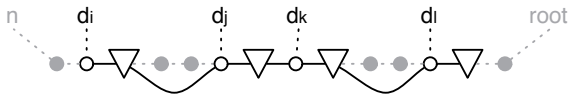


Figure 5.3: The skeleton path of a node n extracted from Figure 5.2

5.4 C-XML data structure

In this section, we specify schema *C-XML* that defines the data structure of C-XML. C-XML extends P-XML with a set of CPAs. CPAs are things that represent viability dependencies in C-XML such that the viability and aliveness of nodes are disconnected from the tree-structure. Additionally, we specify schema $PD_{C\text{-}XML}$ that defines possible documents as an extension to the C-XML schema.

Schema of C-XML data structure Schema *C-XML* defines the data structure of C-XML:

C-XML

$$\begin{aligned}
&P\text{-}XML \\
&cpaSet : \mathbb{P}(NODE \times NODE) \\
&getWSD : NODE \nrightarrow \mathbb{P}(NODE \times NODE) \\
\hline
&cpaSet = possChoices \\
&\mathrm{dom}\, getWSD = ordnodes \\
&\bigcup(getWSD(\!|ordnodes|\!)) \subseteq cpaSet
\end{aligned}$$

Entries in *cpaSet* are called CPAs. A CPA is a tuple of a probability node and a possibility node. CPAs in *C-XML.cpaSet* and possible choices in *P-XML.possChoices* are the same. Function *getWSD* associates nodes with a set of CPAs that describe the set of possible documents of which a node is member. This set of CPAs is called the *world set descriptor* of that node. World set descriptors use solely CPAs that are member of *cpaSet*.

Schema of a possible document Schema $PD_{C\text{-}XML}$ defines possible documents for C-XML. Like the $PD_{P\text{-}XML}$ schema extends *AbsPD*, schema $PD_{C\text{-}XML}$ also extends *AbsPD*.

$PD_{C\text{-}XML}$

$$\begin{aligned}
&AbsPD \text{ but with } C\text{-}XML \text{ instead of } P\text{-}XML \\
\hline
&discardedAllnodes = \{n : NODE \mid n \in ordnodes \land \\
&\quad (\exists (v \mapsto d) : getWSD\ n \bullet v \in \mathrm{dom}\, worldID \ \land\ worldID\ v \neq d) \\
&\quad \bullet\ worldID\ v \neq d) \bullet n\}
\end{aligned}$$

We defined *AbsPD* to extends *P-XML* in order to define possible documents for P-XML with $PD_{P\text{-}XML}$. However, for $PD_{C\text{-}XML}$, we require *AbsPD* to extend *C-XML*. Otherwise, the function *getWSD* could not be used.

A possible document identifier Like in P-XML, a possible document identifier is a set of choices that select for each choice point one alternative. This possible document identifier is the function $PD_{C\text{-}XML}.worldID$. Schema $PD_{C\text{-}XML}$ defines the possible document that is identified with a possible document identifier.

A set of CPAs describes a set of of possible documents A possible document identifier specifies a set of CPAs that identifies one particular possible document encoded by a c-document. Let *wsd* be a set of CPAs that selects for a subset of choice points one alternative. We define *wsd* to describe those possible worlds of which *wsd* is a subset of their possible document identifier. We refer to *wsd* as a world set descriptor (WSD). It holds that a possible document identifier is also a WSD for which holds that the set of possible worlds it refers to contains one entry.

The definition of WSD for P-XML is semantically equivalent to the definition of WSD for C-XML. The only difference is that WSDs for P-XML contain possible choices while WSDs for C-XML contain CPAs.

5.5 P-XML to C-XML mapping

In this section, we define a P-XML to C-XML database mapping $(trans, id)$ where *trans* is a P-XML to C-XML data mapping and *id* is the identity function.

5.5.1 Data mapping

We describe the P-XML to C-XML data mapping *trans* as a replacement of distributional nodes in P-XML for CPAs in CPA. Mapping *trans* does not alter ordinary nodes We have the obligation to show that the same uncertainty distribution is represented by CPAs associated to ordinary nodes as distributional nodes in a tree.

We specify *trans* as follows. Let *pd* be an arbitrary p-document and *cd* be an arbitrary c-document. It holds that $trans(pd) = cd$ if:

- The set of ordinary nodes in *cd* is the same as the set of ordinary nodes in *pd*. Consequently, *pd* and *cd* have the same set of text nodes and XML nodes.
- Edges between ordinary nodes in *pd* are present in *cd*.
- Edges from ordinary nodes to possibility nodes in *pd* are altered to point to the closest ancestor ordinary node –the term 'closest' is discussed in Appendix A.1.
- Distributional nodes in *pd* are not part of *cd*.
- Each edge $(poss \mapsto prob)$ where *poss* is a possibility node and *prob* is a probability node in P-XML is represented as a CPA in C-XML that is associated to all descendant ordinary nodes of *poss*. Consequently, for any n, possible choices that lie on $\Uparrow_n$ are represented as CPAs associated to n such that n is associated with the same set of viable dependencies, however, differently represented.

For illustration purposes, Figure 5.4 shows the result of *trans* applied to our running example in Figure 2.8.

Proof obligation We have the obligation to proof that the same set of possible worlds is represented under *trans*. Proof sketch: In P-XML, for any ordinary node n holds that the aliveness of n is determined by the distributional nodes that lie on $\Uparrow_n$. In C-XML, for any ordinary node n holds that the aliveness n is determined by the CPAs assigned to n. In both data models, viability dependencies are represented as a tuple of a probability node and a child possibility node. The

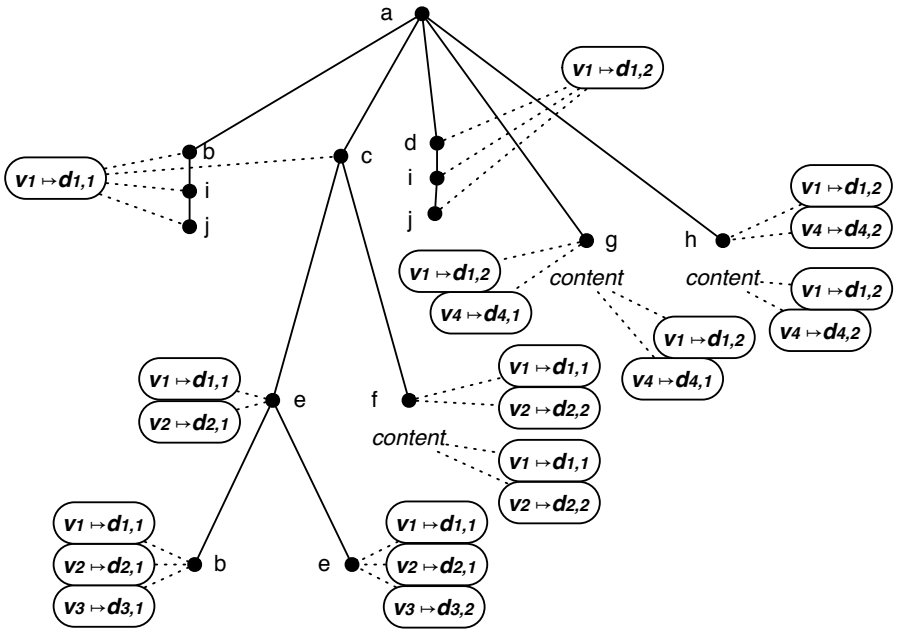


Figure 5.4: The result of *trans* applied to Figure 2.8

semantics of such tuple is in both models the same. Hence, the aliveness of all nodes is preserved under *trans*.

Next, if the aliveness of all nodes is preserved under *trans*, it has to be proven that the same set of possible is represented under *trans*. For P-XML, schema $PD_{P\text{-}XML}$ defines how a possible document is extracted from a p-document in terms of what nodes are part of a possible document given a complete set of choices. For C-XML, schema $PD_{C\text{-}XML}$ defines possible documents in a similar fashion. Given the fact that both data models represent viable dependencies in a similar fashion, such proof should be easy to obtain.

5.5.2 Query mapping

In the data models P-XML and C-XML, encoded possible worlds correspond with actual XML-documents encoded as possible documents. We designed *trans* such that the same set of possible worlds is represented under *trans*. Hence, the same set of XML-documents is encoded under *trans*.

Query evaluation on P-XML data is discussed in the work of Van Keulen et al. [51]. They derive the semantics for query evaluation on P-XML data from the possible worlds model and the semantics of query evaluation on the XML model such that the P-XML model and the XML model support the same query language. A p-document encodes a set of XML-documents for which each instance represents one possible world. The result of a query evaluated on a p-document is a set of possible answers where each answer is obtained as the query evaluated on each encoded XML-document. The same reasoning is applicable to the C-XML model.

Let *pd* be a p-document and *cd* be a c-document such that $cd = trans(pd)$. It follows that *pd* and *cd* encode the same set of XML documents, say *xs*. According to the abovementioned, the result of a query q evaluated on *pd* is a set of possible answers:

$$qe_{pxml}(pd, q) = \{qe_{xml}(x_1, q), \ldots, qe_{xml}(x_n, q)\} \text{ where } xs = \{x_1, \ldots, x_n\}$$

Since *cd* is another encoding for *xs*, the following also holds:

$$qe_{cxml}(cd, q) = \{qe_{xml}(x_1, q), \ldots, qe_{xml}(x_n, q)\} \text{ where } xs = \{x_1, \ldots, x_n\}$$

Since q evaluated on pd results in the same answer as evaluated on cd, we define a query mapping from P-XML to C-XML as the identity function id.

5.6 C-XML into $\mathcal{U}$ mapping

In this section we specify database mapping $(sem_{cxml}, sem_{xpath})$ where data mapping sem_{cxml} maps c-documents to instances of *U-Database* and query mapping sem_{xpath} maps XPath queries to patterns.

5.6.1 Data mapping

C-XML expresses contents in a tree. Such tree is constructed as a set of nodes and a set of edges. A node is annotated with CPAs, a tag label and an element kind. An edge relates tho nodes through the parent/child axis.

We specify sem_{cxml} as follows. Let cd be an arbitrary c-document and ud be an arbitrary *U-Database*. It holds that $ud = sem_{cxml}(cd)$ if:

$$
\begin{array}{l}
\exists\, f_1 : cd.nodes \rightarrowtail\!\!\!\!\!\rightarrow ud.contents \bullet \\
\quad (\forall\, n, n' : cd.nodes \bullet \\
\qquad (n, n') \in cd./\texttt{parent} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{par/child}) \in ud.relSet \wedge \\
\qquad (n, n') \in cd./\texttt{ancestor-or-self} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{anc-or-self/desc-or-self}) \in ud.relSet \wedge \\
\qquad (n, n') \in cd./\texttt{ancestor} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{anc/desc}) \in ud.relSet \wedge \\
\qquad (n, n') \in cd./\texttt{preceding} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{prec/foll}) \in ud.relSet \wedge \\
\qquad (n, n') \in cd./\texttt{preceding-sibling} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{prec-sib/foll-sib}) \in ud.relSet \wedge \\
\qquad (n, n') \in cd./\texttt{sibling} \\
\qquad\qquad \iff (f_1\ n, f_1\ n', \texttt{/sibling}) \in ud.relSet\ \ ) \wedge \\
\quad (\forall\, n : cd.nodes \bullet \\
\qquad n \in cd.xmlnodes \iff (`kind' \leftrightarrow `xml') \in getProp(f_1\ n) \\
\qquad n \in cd.xmlnodes \iff (`tag' \leftrightarrow getTag\ n) \in getProp(f_1\ n) \\
\qquad n \in cd.textnodes \iff (`kind' \leftrightarrow `text') \in getProp(f_1\ n) \\
\qquad n \in cd.textnodes \iff (`text' \leftrightarrow getPCData\ n) \in getProp(f_1\ n)\ \ ) \\
\exists\, f_2 : cd.probnodes \rightarrowtail\!\!\!\!\!\rightarrow first\ (\!|ud.rvaSet|\!) \\
\exists\, f_3 : cd.possnodes \rightarrowtail\!\!\!\!\!\rightarrow second\ (\!|ud.rvaSet|\!) \\
\exists\, f_4 : cd.cpaSet \rightarrowtail\!\!\!\!\!\rightarrow ud.rvaSet \bullet \\
\quad (\forall (v \mapsto d) : cd.cpaSet \bullet \\
\qquad (f_2(v) \mapsto f_3(d)) \in udb.relSet\ \ )
\end{array}
$$

Preserve the concept of a node Bijection f_1 specifies a one to one correspondence between ordinary nodes in cd and objects in ud such that such that each node in a c-document is represented as an object under f_1.

Preserve tree-relationships The query language of C-XML is XPath. We described this language in Section 2.2. XPath specifies a set of axes. An axis is a tree-relationship derived from the parent/child axis.

Formalism $\mathcal{U}_{node}$ is not designed to evaluate tree-relationships derived from the parent/child axis. Therefore, we take the approach to represent XPath axes as directed relations between objects. This is shown in the specification of sem_{cxml} as for each node pair (n, n') that are related

through some axis α are represented as $(f_1\ n, f_1\ n', \alpha') \in ud.relSet$ where α' represents α and its inverse axis.

This approach has one very important advantage. The evaluation of an XPath axis α with $\mathcal{U}$ boils down to the evaluation of pairs of nodes until an entry is found that relates two nodes n and n' through α'. It holds that n and n' are related through α' in those possible worlds of which n and n' are member.

Preserve node properties We consider nodes in C-XML to have the following properties: element type, PCData and node kind. In schema *C-XML*, the first two are captured with the functions *XML.getTag* and *XML.getPCData*. The latter is captured with a partitioning of *allnodes* to $\langle xmlnodes, textnodes \rangle$. The specification of sem_{cxml} preserves node properties as property assignments to the objects that represent these nodes.

Preserve aliveness CPAs and RVAs are semantically equivalent. For $\mathcal{U}$ holds that two RVAs $(rvar \mapsto rval)$ and $(rvar' \mapsto rval')$ conflict if $rvar = rvar'$ and $rval \neq rval$. A WSD that holds conflicting CPAs describe an empty set of possible worlds. The same principle applies to C-XML; a WSD describes an empty set of possible worlds if it contains the CPAs $(v \mapsto d)$ and $(v' \mapsto d')$ for which holds that $v = v'$ and $d \neq d'$. Hence, the bijections f_2, f_3 and f_4 of sem_{cxml} ensure that aliveness is preserved under sem_{cxml}.

Proof obligation We have the obligation to proof that the same set of possible worlds is represented under sem_{cxml}. Proof sketch: In C-XML, for any ordinary node n holds that the aliveness n is determined by the CPAs assigned to n. In U-Rel, for any row r holds that the aliveness r is determined by the RVAs assigned to r. In both data models, viability dependencies are represented as a tuples where the second element is assigned to the first. The semantics of such tuple is in both models the same. Hence, the aliveness of all nodes is preserved under sem_{cxml}.

Next, if the aliveness of is preserved under sem_{cxml}, it has to be proven that the same set of possible is represented under sem_{cxml}. For C-XML, schema $PD_{C\text{-}XML}$ defines how a possible document is extracted from a p-document in terms of what nodes are part of a possible document given a complete set of choices. In order to show that sem_{cxml} preserves the same set of possible worlds, it has to be shows that for each set of choices *wid* that forms a world identifier, the possible world identified by *wid* is the same possible world as identified by $f_4(wid)$.

5.6.2 Query mapping

XML queries typically specify patterns of selection predicates on multiple elements that have some specified tree-relationships. Bruno et al. [10] propose to represent XML queries as twig patterns. A *twig pattern* specifies a required tree-pattern or tree-structure as a twig. A twig is a small tree that uses node-labels to specify desired element types and edges to specify desired parent-child relations or desired ancestor-child relations. The concept of a twig pattern is a widely used concept in research areas that focus on efficient XML query evaluation [37, 11].

In Chapter 4, we introduce the concept of a *Pattern*. This concept specifies a desired database structure in terms of candidates. We capture the semantics of the evaluation of a *Pattern* on a *Database* instance with the concept of a *Match* and the evaluation of a *Pattern* on a *U-Database* with the concept of a *U-Match*.

Basically, patterns of $\mathcal{U}$ have the same semantics as the well known twig patterns. However, twig patterns specify solely the parent/child axis and the ancestor/descendant axis while patterns have support for all XPath axes listed in Table 2.1.

We consider sem_{xpath} from XPath expressions to patterns to be the well known translation of XPath to twig patterns. If it holds that the same set of possible worlds is represented under sem_{cxml}, query mapping sem_{xpath} guarantees that the same question is specified.

For illustration purposes, we visualize an application of sem_{cxml} in Figure 5.5. XPath query q is shown on the top, its corresponding pattern p is listed below. The parts of p that correspond

with q and vice versa are denoted with a $\longleftrightarrow$-sign. Pattern p is constructed of five candidates $c_1, \ldots, c_5$ from left to right. We observe two predicates that each contain a location step. The first predicate makes use of a function *string*() of which the result is compared to the string 'Comedy'. Function *string*() returns the PCData value of a text node that has a parent of the XML type "genre". The equality check has to verify if this value is equal to 'Comedy'. The second predicate "[./directors]" requires movie elements to have a parent of element type "directors". In $\mathcal{U}_{node}$, this boils down to c_4 to have a parent/child-relation with c_5. Since c_5 originates from a predicate, c_4 is the answer to p.

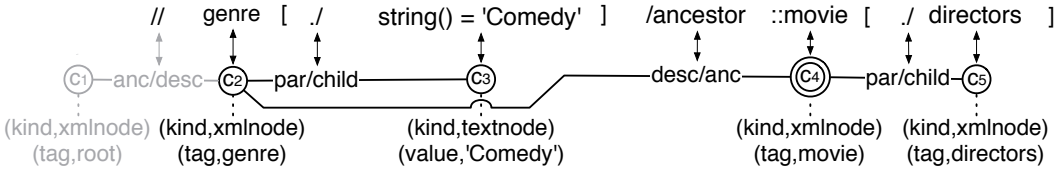


Figure 5.5: From an XPath expression to a pattern

5.7 Summary

This chapter presents our advancing understanding of a P-XML into $\mathcal{U}$ mapping. We construct a P-XML into $\mathcal{U}$ mapping as a sequential composition of *trans* and sem_{cxml}. Mapping *trans* maps p-documents to c-documents. Mapping sem_{cxml} maps c-documents into $\mathcal{U}$.

We map p-documents to c-documents in order to disconnect the representation of uncertainty from the tree-structure. In order to make it plausible that our approach represents the same set of possible worlds under *trans*, we introduce the concept of a viability dependency and the concept of aliveness and give a proof sketch based on these concepts.

We map c-documents into $\mathcal{U}$ in order to represent XPath axes as pairs of nodes. This approach has as main advantage that with solely knowledge about nodes, XPath axes can be evaluated.

Chapter 6

U-Rel expressed in an Abstract Formalism

This chapter presents our advancing understanding to specify a mapping from U-Rel to $\mathcal{U}$. Our only intention is to present our ideas of such mapping. We do not have the ambition to define a U-Rel to $\mathcal{U}$ in detail.

We defined $\mathcal{U}$ in Chapter 4 as the extension of an abstract formalism of a traditional data model with a similar uncertainty management mechanism as U-Rel –the uncertain relational data model of MayBMS. The details of this uncertainty management mechanism are defined in Chapter 4 and are not repeated in this chapter.

Figure 6.1 provides an overview of the contributions of this chapter related to the overall approach to design a correct P-XML into URDBMS mapping.

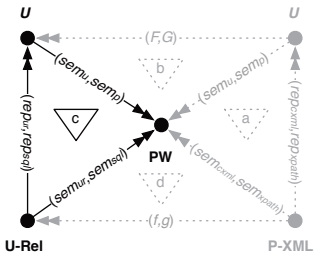


Figure 6.1: Overview of the contributions of Chapter 6

6.1 U-Relational model

Definition of the U-Relational data model We postulate a collection of rows and a collection of columns:

$$[ROW, COLUMN]$$

Schema *U-Rel* defines the data structure of U-Rel –the data model of Antova [5]– as follows:

U-Rel

$$\begin{array}{l} rows : \mathbb{P}\ ROW \\ cols : \mathbb{P}\ COLUMN \\ rowattr : \mathbb{P}\ ROW \times COLUMN \longrightarrow VAL \\ getWSD : ROW \nrightarrow \mathbb{P}(RVAR \times RVAL) \\ rvaSet : \mathbb{P}(RVAR \times RVAL) \\ \hline \bigcup(getWSD(\!|contents|\!)) \subseteq rvaSet \\ \forall\, o : contents; rva, rva' : rvaSet \mid \{rva, rva'\} \subseteq getWSD\ o \bullet \\ \quad first\ rva = first\ rva' \implies second\ rva = second\ rva' \\ \forall\, r, r' : rows \bullet \#getWSD\ r = \#getWSD\ r' \\ \mathrm{dom}\ rowattr = rows \times cols \end{array}$$

A U-Relation is a relational table –a grid data structure defined as a set of rows *rows* and a set of columns *cols*– extended with a function *getWSD* that assigns a world set descriptor (WSD) as

a bag of RVAs to each row. However, for simplicity, we specify the *getWSD* as a function that assigns a set of RVAs to a row. The use of RVAs in U-Rel is semantically equivalent to the use of RVAs in $\mathcal{U}$. Let r be an arbitrary row in *rows*. We consider r as a tuple constructed of row attributes *as* where $as = rowattr(r \times cols)$ such that for each $a_i \in as$ holds that $a_i = rowattr(r, c_i)$ –the intersection of r with $c_i \in cols$.

The main reason why *getWSD* assigns a bag of RVAs instead of a set is that RVAs are represented as triples of columns. We refer to columns that represent RVAs as conditional columns. The semantics of adding a column to a table is appending each row of that table with one row-attribute. Hence, if an U-Relation has i triples of conditional columns, each row is associated with i RVAs.

The left side of Figure 6.2 shows a U-Relations. This U-Relation has the columns 'id', 'pre', 'weight'. Columns with round edges are referred to as *conditional columns*. Each triple of conditional columns defines an RVA for each row. We identify the set of RVAs associated to an individual row is its WSD. Due to the fact that RVAs are represented with triples of columns, all rows in a U-Relation are associated with the same number of RVAs.

Query evaluation We give a simplified description of SQL query evaluation. In order to evaluate an SQL query, all rows in the Cartesian product of tables in the FROM-clause are evaluated. Each row in the Cartesian product that satisfies the conditions specified in the WHERE-clause is a query answer.

If an SQL query is evaluated on U-Relations, the query evaluation process has the extra task to associate each query answer with a WSD that is constructed as a set of RVAs. More specifically, the WSD of a query answer is the union of the WSDs of rows from which that query answer derives. It follows from the definition of the Cartesian product that a query result derives from one row of each table specified in the FROM-clause.

The actual implementation of U-Rel in MayBMS does not construct the WSD of a query answer as a union, but as a concatenation of RVAs. As a consequence, the WSD of query answers may contain duplicates. It holds that the WSD of each query answer is constructed of the same number of RVAs. Hence, the result of an SQL query evaluated on U-Relations is a U-Relation.

The uncertainty management mechanism of U-Rels is similar to $\mathcal{U}$. We refer to Chapter 4 for the semantics and use of RVAs and WSDs.

6.1.1 U-Relations adhere to the possible world semantics

Antova [5] shows that a U-Relation represents a set of possible worlds and query evaluation on U-Relations adheres to the possible worlds semantics. Therefore, we consider the double headed arrow between U-Rel and PW to be proven.

6.2 U-Rel expressed in $\mathcal{U}$

6.2.1 Data mapping

We specify a data mapping sem_{ur} that maps U-Relation to *U-Database*. We design sem_{ur} such that (1) the concept of a row is preserved under f, (2) the concept of a column is preserved under f, and (3) the same set of possible worlds is represented under f.

We specify sem_{ur} as follows. Let *ur* be a c-document and *ud* be a *U-Database*. It holds that

$sem_{ur}(ur) = ud$ if:

$$
\begin{array}{l}
\exists\, f_1 : ur.rows \rightarrowtail\!\!\!\!\!\rightarrow ud.contents; \\
\exists\, f_2 : ur.cols \rightarrowtail\!\!\!\!\!\rightarrow \bigcup \mathrm{dom}(\!|\mathrm{ran}\ getProp|\!); \\
\exists\, f_3 : \mathrm{ran}\ ur.rowattr \rightarrowtail\!\!\!\!\!\rightarrow \bigcup \mathrm{ran}(\!|\mathrm{ran}\ ur.getProp|\!) \bullet \\
\quad (\forall (r, c) : ur.rows \times ur.cols \bullet \\
\qquad rowattr\,(r, c) = f_3\ getProp(f_1\ r)(f_2\ c)\ \ ) \\
\exists\, f_4 : ur.rvaSet \rightarrowtail\!\!\!\!\!\rightarrow ud.rvaSet \bullet \\
\quad (\forall\, r : ur.rows \bullet \\
\qquad getWSD\ r = getWSD\,(f_1\ r)\ \ )
\end{array}
$$

We specify sem_{ur} as four bijections. Bijection f_1 represents each row in $ur.rows$ as an object in $ud.contents$. Bijection f_2 represents each column in $ur.cols$ as a property of ud. Bijection f_3 ensures that each row-attribute value a is represented as property assignment $ur.getProp(f_1\ r)(f_2\ c)$ where $a = ur.rowattr(r, c)$. Last, bijection f_4 ensures that the object that represents a row is associated with the same set of RVAs. It follows from the specification of sem_{ur} that *U-Database* instances derived from sem_{ur} do not contain relations between objects.

We apply sem_{ur} to the U-Relations in Figure 6.2. The result is a U-Database instance. We refer to the U-Relation as ur and we refer to the U-Database instance as ud. Each row in ur is represented as an object –denoted as a black circle– in ud. Row-attributes are represented as property-assignment. More specifically, each object is has one property assignment per column in ur. For each row and its object counterpart holds that their WSD is constructed of the same set of RVAs.

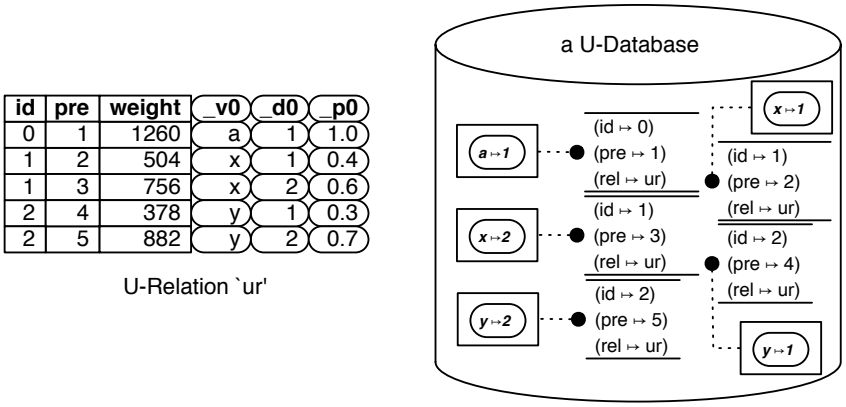


Figure 6.2: U-Rel to $\mathcal{U}$ data mapping

6.2.2 Query mapping

We specify sem_{sql} –a query mapping from SQL to patterns– as follows. Let sql be an arbitrary SQL query and p be an arbitrary pattern. It holds that $sem_{sql}(sql) = p$ if:

- There exists a bijection from U-Relations specified in the FROM-clause of sql to candidates in p such that for each U-Relation ur and candidate $sem_{sql}(ur)$ holds that a required property is assigned to $sem_{sql}(ur)$ of the form $(\text{`}rel' \mapsto ur.name)$ where $ur.name$ is the relation name of ur.

- There exists a bijection from constraints specified in the WHERE-clause of sql to required properties in p. Constraints are of the form (c, op, c') where c is a column of a U-Relation specified in the FROM-clause of sql, op is a boolean operator and c' is a value or a column of

a U-Relation. Let function $getT$ obtain the U-Relation of a column. If c' is a value and op is equality operator, we assign property $(f_2\ c \mapsto c')$ to candidate $sem_{sql}(getT\ c)$ where f_2 is the same bijection as used for the data mapping. If c' is a column, we define a property expression that relates candidate $sem_{sql}(getT\ c)$ to $sem_{sql}(getT\ c')$ with the expression $(f_2\ c, f_2\ c', c\ op\ c')$. Property expressions on candidates are discussed in Section 4.2.

For illustration purposes, we show a simple mapping of an SQL query in Figure 6.3. The FROM-clause specifies $t1, t2$ and $t3$ as abbreviations for table ur. Table ur is illustrated in Figure 6.2. We represent these three abbreviations as the three candidates $\{c_1, c_2, c_3\}$ that all are associated with the property assignment $(rel \mapsto `ur')$. We count 3 conditions in the WHERE-clause. The first condition requires column $t1.pre$ to have a fixed value. We represent this conditions as property assignment $(pre \mapsto 2)$ to candidate c_1 that represents $t1$. The other two conditions of the WHERE-clause involve two columns. We map condition $t1.id = t2.id$ to $(c_1, c_2, c_1.id = c_2.id)$ where c_1 represents t_1 and c_2 represents t_2. Analogously, we map $t2.pre = t3.pre$ to $(c_2, c_3, c_2.pre = c_3.pre)$.

We have the obligation to proof that the same question is specified under sem_{sql}.

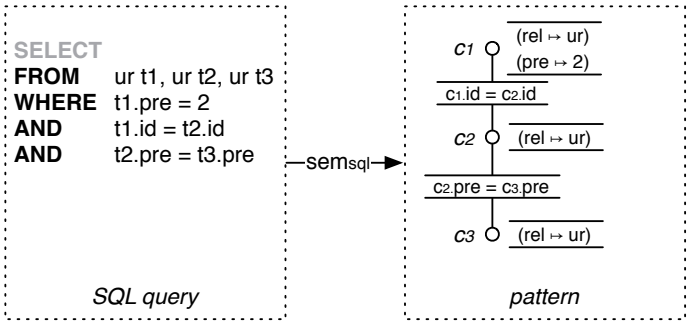


Figure 6.3: An SQL query mapped to a pattern

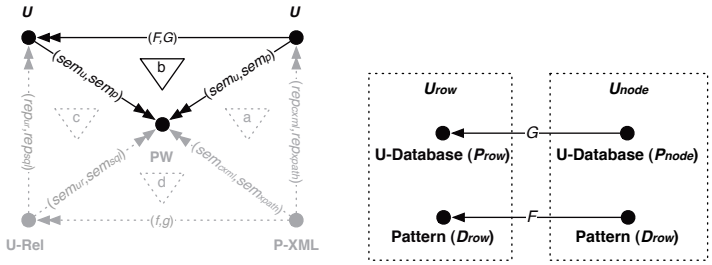
Chapter 7

Mapping of Nodes to Rows

This chapter presents our advancing understanding to specify a mapping from $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$. Our only intention is to present our ideas of such mapping. We do not have the ambition to define a $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ in detail.

We defined $\mathcal{U}$ in Chapter 4 as the extension of an abstract formalism of a traditional data model in Chapter 4. We write $\mathcal{U}_{node}$ instead of $\mathcal{U}$ to state that $\mathcal{U}$ represents a tree-structured data model. Analogously, $\mathcal{U}_{row}$ represents a table-structured data model.

Figure 7.1a provides an overview of the contributions of this chapter related to the overall approach to design a correct P-XML into URDBMS mapping.



(a) (b)

Figure 7.1: Overview of the contributions of Chapter 7

7.1 Data mapping

Schema *U-Database* ($\mathcal{D}$) –introduced in Section 4.5– defines the data structure of $\mathcal{U}$ as a graph with objects, relations and property assignments. We write $\mathcal{U}_{node}$ to denote $\mathcal{U}$ that represents a tree-structured data model. Analogously, we write $\mathcal{U}_{row}$ to denote $\mathcal{U}$ that represents a table-structured data model.

Application of the XPath Accelerator encoding Data structures $\mathcal{D}_{node}$ and $\mathcal{D}_{row}$ are very similar. Both are graph data structures with objects associated with property assignments and RVAs. However, $\mathcal{D}_{row}$ does not support relations between objects. In $\mathcal{D}_{node}$, relations between objects represent tree-relations in P-XML. In order to represent a tree-structure in $\mathcal{D}_{row}$, we borrow the work of Grust et al. [21]. They show that all tree-relationships are preserved for an element encoding that uses the pre-order rank and size of nodes.

Specification of F We specify F –a data mapping from $\mathcal{D}_{node}$ to $\mathcal{D}_{row}$– as two steps. In the first step, each object in $\mathcal{D}_{node}$ is associated with two property assignments that capture pre-order rank and size. Both properties are in more detail described in Section 2.3.2. In the second step, relations between objects are discarded. This operation does not result in a loss of information, since the pre-order rank and size allow for the evaluation of XPath axes with the range conditions specified in Section 2.3.

Same set of possible worlds The specification of F does not alter objects and world set descriptors. It follows from Axiom 1 that the same set of possible worlds is represented under F.

7.2 Query mapping

Data model $\mathcal{U}$ uses the concept of a *Pattern* ($\mathcal{P}$) as queries. Instances of $\mathcal{P}_{node}$ are constructed of (1) candidates that represent required nodes, (2) required values for the properties *kind*, *tag* and *value*, and (3) relations between candidates that represent required relations between nodes.

Specification of G Data mapping F maps objects that represent nodes to objects that represent rows. We specify G to use the same set of candidates such that the interpretation of these candidates changes from a set of required nodes to a set of required rows. Next, the properties *kind*, *tag* and *values* are preserved under F such that the row that represents a node is associated with the same property assignments. Therefore, candidates associated with properties that use properties *kind*, *tag* or *values* are unaltered. Last, F annotate objects that represents nodes with a *pre* property value and a *size* property value such that tree-relations between any two nodes are evaluated as a range condition as specified in 2.3.2. Hence, G represents a required relations between objects as a property expression that requests for a range condition to hold. This range conditions checks for the presence of the required relation it represents.

Part III

Design

Chapter 8

P-XML into URDBMS Data Mapping

In this chapter, we give a design for P-XML to U-Rel data mapping f that is specified in Part II. Figure 8.1 shows that $f = f' \circ \mathit{trans}$ where *trans* maps p-documents –document instances of P-XML– to c-documents –document instances of C-XML– and f' maps c-documents to U-Relations –database instances of U-Rel. We design f such that its behaviour is semantically equivalent to $f' \circ \mathit{trans}$.

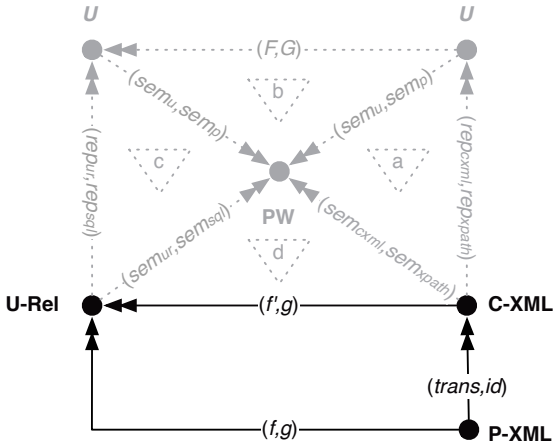


Figure 8.1: Data mapping f is constructed as $f' \circ \mathit{trans}$

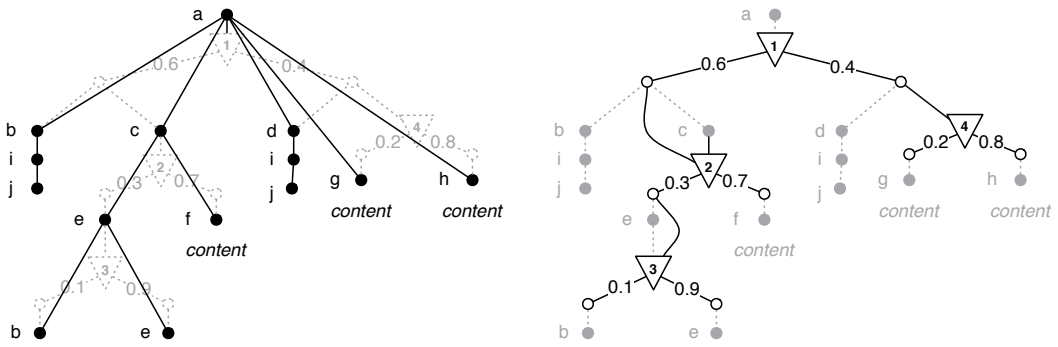
In Section 8.1, we design *trans*, a P-XML to C-XML mapping that transforms p-documents to c-documents. This mapping consists of a breakdown of a p-document into two parts –the skeleton and the flesh– which are merged afterwards into a c-document. We refer to this merge process as *gluing*. In Section 8.2, we design $f' \circ \mathit{trans}$ as a mapping of the skeleton and the flesh into a URDBMS in order to perform a glue process with U-Relations. In Sections 8.3 and 8.4, we discuss the details of mapping the flesh and mapping the skeleton into a URDBMS. We leave the details of a glue process carried out by a URDBMS for future chapters.

8.1 Construct a c-document from flesh & skeleton

In order to construct a c-document from a p-document, we introduce the terminology skeleton and flesh. The *skeleton* is a p-document stripped of all its ordinary nodes. The *flesh* is a p-document stripped of all its distributional nodes. As such, the flesh an ordinary XML-document. Edges in the original p-document that point to discarded nodes are pointed to the closest non-discarded node in the flesh or skeleton.

For illustration purposes, we extract the flesh and skeleton from the p-document in Figure 2.8 in order to retrieve the c-document in Figure 5.4. The result of the flesh extraction is found in Figure 8.2a. Analogously, the result of the skeleton extraction is found in Figure 8.2b.

We introduce trans_{fl} to denote the extraction of the flesh from a p-document and trans_{sk} to denote the extraction of the skeleton from a p-document. We define the *P-XML* to *C-XML* mapping *trans* as follows:



(a) Flesh representation

(b) Skeleton representation

Figure 8.2: Probabilistic XML building blocks

$$
\begin{array}{l}
trans : \textit{P-XML} \longrightarrow \textit{C-XML} \\
trans_{fl} : \textit{P-XML} \longrightarrow \textit{XML} \\
trans_{sk} : \textit{P-XML} \longrightarrow \mathbb{P}\ \textit{CPA} \\
glue_{pxml} : \textit{XML} \times \mathbb{P}\ \textit{CPA} \longrightarrow \textit{C-XML} \\
\hline
\forall\, pd : \textit{PXML} \bullet trans\ pd = glue_{pxml}(trans_{fl}(pd), trans_{sk}(pd))
\end{array}
$$

For a p-document pd, we define *trans* to construct a c-document that represents the same set of possible worlds as pd. Mapping *trans* is composed of three parts. It merges the results of $trans_{fl}$ and $trans_{sk}$ with a glue process –denoted as $glue_{pxml}$. The result of $trans_{sk}$ is a set of CPAs. A CPA is an edge from a possibility node to a possibility node. Since the skeleton is a tree constructed of solely distributional nodes, we interpret the skeleton as a set of CPA. Function $glue_{pxml}$ assigns CPAs to nodes such that for each node n in the flesh, all possible choices on the skeleton path $\Uparrow n$ are represented as CPAs assigned to n.

8.2 First design of P-XML into URDBMS mapping from flesh & skeleton

In the previous section, we define *trans*, a *P-XML* to *C-XML* mapping that is based on the functions $trans_{fl}$ and $trans_{sk}$ that extract the flesh and the skeleton from a p-document. In this section, we introduce the functions f_{fl} and f_{sk} that map flesh and skeleton into a URDBMS in order to let a URDBMS conduct a relational glue process:

$$
\begin{array}{l}
f : \textit{P-XML} \longrightarrow \textit{U-Rel} \\
f_{fl} : \textit{XML} \longrightarrow \mathbb{P}\ \textit{REL} \\
f_{sk} : \mathbb{P}\ \textit{CPA} \longrightarrow \mathbb{P}\ \textit{RVA} \\
glue_{rel} : \mathbb{P}\ \textit{REL} \times \mathbb{P}\ \textit{RVA} \longrightarrow \textit{U-Rel} \\
\hline
\forall\, pd : \textit{PXML} \bullet f(pd) = glue_{rel}(f_{fl}(trans_{fl}(pd)), f_{sk}(trans_{sk}(pd)))
\end{array}
$$

We consider a glue process that merges the result of f_{fl} and f_{sk} as our first design of an P-XML into URDBMS mapping. Our design is constructed of several parts. The parts $trans_{fl}$ and $trans_{sk}$ are introduced in the previous section. Function f_{fl} maps the flesh into a URDBMS as a set of relational tables. Function f_{sk} maps the skeleton into a URDBMS as a set of RVAs. A relational glue process assigns RVAs to the relational tables such that the result is a set of U-Relations.

Note that the type of f_{fl} indicates that f_{fl} is no more than a traditional XML into RDBMS mapping for which many approaches exist. Also note that the mapping of the skeleton is a mapping

from CPAs to RVAs. Since CPAs are semantically equivalent to RVAs, a mapping between the two is straightforward. We define this mapping in the following section.

In order to show that our design conforms to the specification of Part II, we have the obligation to prove that for any p-document pd the following holds:

$$glue_{rel}(f_{fl}(trans_{fl}(pd)), f_{sk}(trans_{sk}(pd))) = f'(trans\ pd)$$

Such proof suffices, because $f'(trans\ pd) = f(pd)$.

8.3 Mapping of the skeleton into a URDBMS

We define f_{sk}, a skeleton mapping that maps CPAs to RVAs as follows. Let *cpaSet* be a set of CPAs and *rvaSet* be a set of RVAs. It holds that $f_{sk}(cpaSet) = rvaSet$ if

$$\exists f_1 : first(\!|cpaSet|\!) \rightarrowtail\!\!\!\!\!\rightarrow first(\!|rvaSet|\!)$$
$$\exists f_2 : second(\!|cpaSet|\!) \rightarrowtail\!\!\!\!\!\rightarrow second(\!|rvaSet|\!)$$

Recall that we defined CPAs as XML-like RVA. The only difference is that CPAs use distributional nodes to represent choices while RVAs use random variable assignments to do the same. We define f_{sk} to represent a choice point v as a random variable $f_1\ v$ and its set of alternatives ds as $f_2(\!|ds|\!)$. It follows that a CPA $(v \mapsto d)$ is represented as the RVA $(f_1\ v \mapsto f_2\ d)$ that denotes the assignment of value $f_2\ d$ to random variable $f_1\ v$.

8.4 Mapping of the flesh into a URDBMS

In this section, we define f_{fl}, a flesh mapping that maps ordinary nodes to rows. We construct f_{fl} as an ordinary XML into RDBMS mapping. However, there are certain requirements that have to be satisfied by such mapping in order to be as operand for a relational glue process. We discuss these requirements in Section 8.4.1

In Section 8.4.2, we introduce Abstract Shared Inlining with XA as parameter (ASI[XA]), a new XML into RDBMS mapping that combines the XA appoach discussed in Section 2.3.2 with the SI approach discussed in Section 2.3.1. Mapping ASI[XA] satisfies all requirements to be used as flesh mapping. Furthermore, ASI[XA] has some interesting properties that fit well with the relational glue process. One important note: in future chapters, we use the XA mapping for examples, because the result of XA fits nicely in two tables, while the result of ASI[XA] is a set of tables.

8.4.1 Flesh mapping requirements

We specify the following three requirements for an XML into RDBMS to be used as flesh mapping.

- Nodes have to be represented as rows. This requirement follows from our approach to derive a P-XML into URDBMS mapping from a $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ mapping –introduced in Chapter 7– where we map objects that represent nodes to objects that represent rows. Hence, a P-XML to URDBMS mapping derived from a $\mathcal{U}_{node}$ to $\mathcal{U}_{row}$ mapping should represent nodes as rows.
- In order for the relational glue process $glue_{rel}$ to assign RVAs to rows, we require that an XML into RDBMS mapping is able to represent different kinds of nodes such that some notion of distributional nodes exists in a URDBMS.
- An XML into RDBMS is required to support the ancestor/descendant axis. This requirement follows from the fact that distributional nodes annotate their descendants. In order to preserve the aliveness of nodes under a P-XML to URDBMS mapping, a node has to be related to distributional nodes that reside in its ancestor axis.

While the flesh does not contain distributional nodes, a glue process has to be able to associate rows derived from the flesh mapping to RVAs that derive from the skeleton mapping. The latter two requirements make this possible.

8.4.2 ASI[XA], a new XML into RDBMS mapping designed as flesh mapping

In this section, we specify ASI[XA], an XML into RDBMS mapping designed for a flesh mapping. This mapping uses the relational data structure and the inline principle of Shared Inlining (SI) in combination with the element encoding of XPath Accelerator (XA) at the cost of using a schema that describes the structure of the flesh. SI and XA are discussed in Sections 2.3.1 and 2.3.2. In this section, we use the terminology of these sections to specify ASI[XA].

The main motivation why we use ASI[XA] over XA as flesh mapping is that the efficiently of $glue_{rel}$ is enhanced for a partitioning of the flesh. We discuss this enhancement in more detail in Section 11.2. An XML into RDBMS has to satisfy the requirements specified in previous section in order to be used as flesh mapping. ASI[XA] does not satisfy the first requirement; the inline principle of ASI[XA] stores multiple nodes in the same row. We deal with this problem in Section 10.3.1.

We describe two topics in this section. The first topic is a data dictionary that enables a more efficient collaboration between ASI[ENC] and XA by an optimization of the data encoding of XA. The second topic is the resulting data encoding of ASI[XA].

columns	*row*
a_pre	1
a_size	24
a_par	-1
a_c_pre	6
a_c_size	9
a_d_pre	17
a_d_size	2
a_c_f_pre	14
a_g_pre	21
a_h_pre	24
a_c_f_pcdata	*content*
a_g_pcdata	*content*
a_h_pcdata	*content*
eT of node kind *a*	

columns	*row*	*row*
b_pre	3	10
b_size	2	0
b_par	1	8
eT of node kind *b*		

columns	*row*	*row*
i_pre	4	18
i_par	3	17
j_pre	5	19
eT of node kind *i*		

columns	*row*	*row*
e_pre	8	12
e_size	4	0
e_par	6	8
eT of node kind *b*		

poss_pre	*poss_size*
0	25
2	13
7	5
9	1
11	1
13	2
16	9
20	2
23	2
possibility nodes	

Figure 8.3: ASI[XA] approach applied to Figure 2.8 with traditional inline rules

8.4.2.1 Data Dictionary

The main ingredient to turn ASI[XA] into a powerful and efficient data structure is a *data dictionary*. A data dictionary is a centralized repository of information about data. We use a data dictionary for a variety of purposes:

1. The relational data structure of ASI[XA] assigns an element table to each element type. A data dictionary provides the means to locate element tables of specific element types. As a consequence, the tag-attribute of the default XA encoding is made redundant.
2. We use a data dictionary to track what node kind is stored in which column(s). As a consequence, the kind-attribute of the default XA encoding is also made redundant.
3. A data dictionary captures what element types are related to other element types through the parent/child axis. This information is specified in the schema of a mapped p-document.

We use this information to evaluate location steps that do not specify a node test. More specifically, if no node test is specified, the query engine cannot determine which element tables have possible relevant XML nodes. It would be costly to search in all element tables. A more efficient approach is to verify what element types qualify to have possible relevant XML nodes. One approach to accomplish this is to let a data dictionary manage which element types are related to each other through the parent/child axis.

4. A data dictionary keeps track of which element types are inlined. This knowledge is used to evaluate the parent/child axis more efficiently.

For illustration purposes, Figures 8.4a and 8.4b present the data dictionary that corresponds with Figure 8.3. Table 'DD_elems' in Figure 8.4a uses column 'type' to store element types, column 'self_id' to associate each unique element type with an identifier, column 'has_pcdata' know for each element type if it has PCData, column 'is_master' to manage if an element type is the master of the element table it resides in, column 'depth' to manage the depth of its element table. Table 'DD_relations' in Figure 8.4b uses column 'table' to store the table of an element type, column 'column' to store the column that holds the pre-order of an element type, column 'self_id' to store the identifier of an element type, column 'par_id' to store pointers to parent element types. Tables 'DD_elems' and 'DD_relations' are used during the mapping of an XPath query to a corresponding SQL query.

type	self_id	has_pcdata	ismaster	depth
a	1	f	t	0
b	2	f	t	3
c	3	f	f	1
d	4	f	f	1
e	5	f	t	3
f	6	t	f	2
g	7	t	f	2
h	8	t	f	2
i	9	f	t	1
j	10	f	f	1

(a) Table 'DD_elems'

table	column	self_id	par_id
eT_a	a_pre	1	-1
eT_b	b_pre	2	5
eT_b	b_pre	2	1
eT_a	a_c_pre	3	1
eT_a	a_d_pre	4	1
eT_e	e_pre	5	3
eT_e	e_pre	5	5
eT_a	a_c_f_pre	6	3
eT_a	a_g_pre	7	1
eT_a	a_h_pre	8	1
eT_i	i_pre	9	4
eT_i	i_pre	9	2
eT_i	i_j_pre	10	9
eT_a	a_c_f_pcdata	-1	6
eT_a	a_g_pcdata	-1	7
eT_a	a_h_pcdata	-1	8

(b) Table 'DD_relations'

Figure 8.4: The data dictionary that corresponds with Figure 8.3

8.4.2.2 Element Encoding

Overview of optimizations for the XA element encoding The original element encoding of XA represents a node n as:

$$n \mapsto \langle pre(n), size(n), level(n), tag(n), kind(n) \rangle$$

However, ASI[XA] uses an optimized encoding:

- A data dictionary makes the tag-attribute and kind-attribute of the XA encoding redundant.
- The level-attribute of the XA encoding is made redundant with following. In Section 2.3.2, we specify the range conditions used to evaluate XPath axes with the XA encoding. Observe that the level-attribute is solely used to evaluate the parent/child axis. In Section A.1, we give a proof to evaluate the parent axis as the evaluation of the closest node. The evaluation of the closest node requires the XA encoding to include the pre-attribute and the size-attribute, but not the level-attribute.

- The size-column of the default XA encoding is solely used to evaluate ancestor/descendant-relations. More precisely, the size-column is used by the ancestor element type. In case all elements of a specific ancestor element type do not have descendants that reside in another element table, we can determine its descendants without a size-column.

Sibling axis support As far as we can tell, element encoding of XA is not suitable to evaluate the sibling axis. In order to support the evaluation of these three axes, we use a parent-reference for a master element type such that each instance of a master element type has a pointer to the pre-order of its parent node. This allows us to evaluate the previously stated XPath axes as follows. Let c be a context node and n be an arbitrary node in an XML-document:

$$\begin{array}{lcl} n \in c\texttt{/sibling} & \Leftrightarrow & par(n) = par(c) \\ n \in c\texttt{/preceding-sibling} & \Leftrightarrow & par(n) = par(c) \wedge pre(n) < pre(v) \\ n \in c\texttt{/following-sibling} & \Leftrightarrow & par(n) = par(c) \wedge pre(n) > pre(v) \end{array}$$

Optimized element encoding of $ASI[XA]$ We summarize the element encoding of ASI[XA] as follows:

- If a node n is (1) an instance of master element type N and (2) N has instances that are not leave node, we represent n as: $\langle pre\ n, size\ n, par\ n \rangle$.
- If a node n is (1) an instance of master element type N and (2) all instances of N are leave nodes, we represent n as: $\langle pre\ n, par\ n \rangle$.
- If a node n is (1) an instance of slave element type N and (2) N has instances that are not leave node, we represent n as: $\langle pre\ n, size\ n \rangle$.
- If a node n is (1) an instance of slave element type N and (2) all instances of N are leave nodes, we represent n as: $\langle pre\ n \rangle$.

For illustration purposes, we apply the ASI[XA] approach to our running example in Figure 2.8. The result is found in Figure 8.3. This figure shows four element tables (eT). We assign a prefix to each attribute that shows host-guest relationships between element types. For example, column 'a_c_f_pre' stores pre-orders for nodes of element kind f. The prefix $a\_c\_$ denotes that element type f is inlined with element type c which itself is inlined with element type a. Observe that the size-attribute is only used for element kinds that have descendants that reside in different element tables. Furthermore, each instance of a master element type has a parent-reference while non-masters do not. For the purpose of completeness, we include a grey table that represents possibility nodes as tuples of pre-order and size. Note that this table is not created by the flesh mapping. We show this table to show that ASI[XA] takes distributional nodes[1] into account.

8.5 Summary

This chapter introduces the concepts flesh and skeleton. The flesh is a collection of ordinary nodes represented as a tree. Analogously, the skeleton is a collection of distributional nodes that we represent as CPAs. CPAs are edges from a possibility node to a probability node.

We use the concepts flesh and skeleton to define f, a data mapping from P-XML to U-Rel. Mapping f (1) extracts the flesh and the skeleton from a p-document, (2) maps the flesh and the skeleton into a URDBMS with flesh mapping f_{fl} and skeleton mapping f_{sk}, and (3) performs a relational glue process in order to merge the results.

We design f_{sk} as a bijection from CPAs to RVAs and f_{fl} as ASI[XA], an XML into RDBMS mapping composed of two other XML into RDBMS mappings. We design a relational glue process in Chapter 10.

[1] Probability nodes do not have to be numbered, because these nodes are by definition parent nodes of possibility nodes. Their occurrences follow from occurrences of possibility nodes.

Chapter 9

Introduction to Relational Gluing

In this chapter, we present the tools to perform a relational glue process. The purpose of a glue process is to associate rows with a WSD such that for each node n represented as row r, the world set of n is described by the WSD of r. A glue process is an iterative process. Iterations of a glue process are referred to as phases. Phases are introduced in Section 9.1 as a merge process of rows with an RVA. Phases operate on rows derived from f_{fl} –a mapping of the flesh– and RVAs derived from f_{sk} –a mapping of the skeleton. We construct a WSD for a row as a set of RVAs. In order to match RVAs with rows, we introduce dependency handles in Section 9.2. Dependency handles provide information to match RVAs to rows. The actual process of assigning RVAs to rows is performed by the G-Join statement. This statement is introduced in Section 9.3. The design of G-Join is required to associate each row with exactly one RVA. In some scenarios, this is not possible. In these scenarios, we associate rows with the don't care choice; a dummy RVA that does not alter the WSD of a row. We discuss the design of the don't care choice in Section 9.4. In Section 9.5, we introduce depth, a property that we use to determine the number of phases a glue process has to perform in order to complete.

9.1 Phases

A mapping from one uncertain data structure to another makes use of some kind of glue process. A *glue process* is a step-wise approach that reconstructs viable dependencies in order to construct a representation of a set of possible worlds. A step or iteration in a glue process is referred to as a *phase*. In one single phase, a set of viable dependencies is reconstructed. The intermediate result of a phase is referred to as a *phase table*. Phase tables are U-Relations for which a subset of viable dependencies is reconstructed. Figure 9.1 provides a sketch of a glue process. Function *glue* denotes a specific operation that takes a phase table as input and generates a new phase table as output through the use of a set of RVAs denoted as skeleton. We identify this operation as the *G-Join* statement –denoted as $G_{\bowtie}$– that applies a *glue method* to the input phase table and skeleton.

A glue process starts with phase table ph_0 instantiated as the flesh. In the first phase, *glue* transforms phase table ph_0 to phase table ph_1. Following phase tables are constructed likewise. Phase table ph_n represents the same set of possible worlds as the original document from which the skeleton and the flesh are derived. If the flesh is represented as a set of tables, a glue process has to be applied to each table in order to reconstruct all viable dependencies.

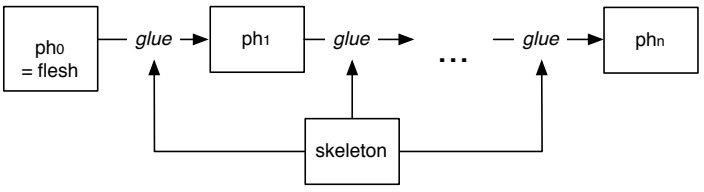


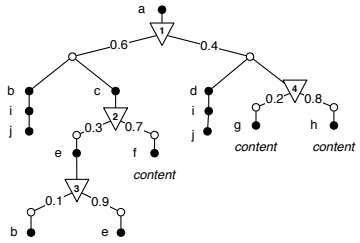
Figure 9.1: A glue process is a step-wise approach

9.2 Dependency handles

In order to reconstruct viable dependencies, we require information that describes which entries in the skeleton should be assigned to what entries in a phase table. We introduce the terminology

dependency handles to refer to this kind of information. Each entry in a phase table is associated with a right dependency handle and each entry in the skeleton is associated with a left dependency handle such that each entry in a phase table is matched with exactly one entry in the skeleton.

For illustration purposes, Figure 9.2b shows a relational table that represents the flesh of Figure 9.2a –a larger copy is found in Figure 2.8– mapped into a URDBMS with the default XA approach discussed in Section 2.3.2. Figure 9.2c shows a set of RVAs that represent the skeleton of Figure 9.2a. The last entry in Figure 9.2c represents the don't care choice that we discuss in Section 9.4. Dependency handles are denoted with triangles. *Right dependency handles* –triangles that point to the right– match *left dependency handles* –triangles that point to the left. Right dependency handles are used in the skeleton and in the flesh and left dependency handles are only used in the skeleton.



(a) P-XML Tree

pre	size	level	tag	kind
1	24	1	'a'	elem
3	2	2	'b'	elem
4	1	3	'i'	elem
5	0	4	'j'	elem
6	9	2	'c'	elem
8	4	3	'e'	elem
10	0	4	'b'	elem
12	0	4	'e'	elem
14	1	3	'f'	elem
15	0	4	-	pcdata
17	2	2	'd'	elem
18	1	3	'i'	elem
19	0	4	'j'	elem
21	1	2	'g'	elem
22	0	3	-	pcdata
24	1	2	'h'	elem
25	0	3	-	pcdata

(b) Flesh mapped into an (U)RDBMS

var_1	val_1
v_1	$d_{1,1}$
v_1	$d_{1,2}$
v_2	$d_{2,1}$
v_2	$d_{2,2}$
v_3	$d_{3,1}$
v_3	$d_{3,2}$
v_4	$d_{4,1}$
v_4	$d_{4,2}$
$v_\star$	$d_\star$

(c) Skeleton mapped to RVAs

Figure 9.2: Dependency handles

Dependency handles describe which entries in the skeleton and form a match. The actual matching is done by a glue method. All glue methods have in common the matching process is based on relating descendants with ancestors.

9.3 G-Join

In this section, we introduce the G-Join statement that performs one phase of a relational glue process. For a set of rows, G-Join associates one RVA to each row. Each following phase, another RVA is associated to each row. At completion of a relational glue process, each row r is associated with a set of RVAs that describes the world set of the node represented by r.

We design G-Join such that no random variable is used twice in the WSD of a row. For example, if the WSD of a row is defined as $\{(v_1 \mapsto d_{1,2}), (v_2 \mapsto d_{2,1})\}$, than a G-join may not add another RVA that involves v_1 or v_2. As a result, G-Join never adds a conflicting RVA to a row. Hence, the cardinality of the input of G-Join is equal to the cardinality of the output.

G-Join has three parameters: (1) a set of rows (represented as a U-Relation), (2) a set of RVAs (represented as a U-Relation), and (3) a join condition that represents a glue method. We consider function *glue* in Figure 9.1 to be the G-Join statement with phase table *ph* as its first parameter, the skeleton represented as a U-Relation *sk* as its second parameter, and a glue method m as its third parameter.

$$
\begin{array}{|l}
G_{\bowtie} : \textit{U-Rel} \times \textit{U-Rel} \times (\textit{ROW} \longrightarrow \textit{RVAR} \times \textit{RVAL}) \longrightarrow \textit{U-Rel} \\
\hline
\textit{res} = G_{\bowtie}\,(\textit{ph}, \textit{sk}, m) \iff \\
\qquad\qquad (\forall\, r : \textit{res.rows} \bullet \\
\qquad\qquad\quad m\, r \in \textit{sk.rvaSet} \wedge \\
\qquad\qquad\quad \textit{res.getWSD}\, r = \textit{ph.getWSD}\, r \cup \{m\, r\} \wedge \\
\qquad\qquad\quad \textit{first}(m\, r) \notin \textit{first}(\!|\textit{ph.getWSD}\, r|\!) \wedge \\
\qquad\qquad) \wedge \\
\qquad\qquad \textit{res.rows} = \textit{ph.rows} \wedge \\
\qquad\qquad \textit{res.cols} = \textit{ph.cols} \wedge \\
\qquad\qquad \textit{res.rowattr} = \textit{ph.rowattr} \wedge \\
\qquad\qquad \textit{res.rvaSet} = \textit{ph.rvaSet} \cup m(\!|\textit{res.rows}|\!)
\end{array}
$$

For each r in the result *res* of G-Join, it holds that (1) $m\,r$ returns an RVA that is part of *sk.rvaSet*, (2) the new WSD of r is constructed as its old WSD with the addition of $m\,r$, and (3) the random variable of $m\,r$ is part of any other RVA in the old WSD of r. Columns, rows and row attributes are similar in *ph* and *res*[1]. The set *res.rvaSet* is defined as a union of *ph.rvaSet* and the RVAs assigned to rows in *res.rows*.

The aliveness of rows in *ph* is changed as a result of G-Join. This follows from the fact that we defined the aliveness of a row r as $r \hat{\|} \textit{getWSD}\, r$. This statement has to hold before and after execution of the G-Join statement. It follows that the effect of $G_{\bowtie}(\textit{ph}, \textit{sk}, m)$ on a row $r \in \textit{ph.rows}$ is: $r \parallel m(r)$.

9.4 Don't care choice

U-Rel –defined in Section 6.1– has one important drawback: rows in U-Relations –instances of U-Rel are bound to have an equal size world set descriptor. In other words, each row of a U-Relation has to be associated with the same amount of RVAs. This requirement is shown by the second last row of schema *U-Rel* and follows from the design choice of MayBMS to represent RVAs as a triple of conditional columns. As a consequence, $G_{\bowtie}$ is required to assign one RVA to each row in a U-Relation, even if no such RVA exists.

Don't care choice provides solution This drawback is dealt with through the use of the *don't care choice*. The *don't care choice* is a added to a p-document during the data mapping process and denoted as $(v_* \mapsto d_*)$ where v_* is a choice point and d_* is the only alternative that can be associated to v_*. Normally, a choice point denotes a place in a p-document where an alternative has to be selected among others. The only exception is a choice point with one alternative; that alternative is certain to be selected. If we place the don't care point at the root of a p-document, it has to be part of all the encoded possible documents.

Relational representation of don't care choice In U-Rel, the don't care choice is represented as an RVA $(\textit{rvar} \mapsto \textit{rval})_*$ where *rval* is the only value that can be assigned to *rvar*. In other words, there does not exists another RVA that assigns a value to *rvar*. We refer to the RVA representation of the don't care choice as the *don't care RVA*.

Use of relation don't care choice In situations in which a row is obliged to take part in a G-Join while it has no RVA to join with, that row is joined with the don't care RVA. The don't care RVA is allowed to be added multiple times to the same row and constitutes the only exception to the second requirement of a G-Join. During the skeleton mapping, the don't care choice is mapped to the don't care RVA. To avoid any misinterpretations, we do not acknowledge (v_*, d_*) to be a viable dependency. Otherwise, any node n mapped into a URDBMS should be associated

[1] In practice, G-Join adds three conditional columns to represent appended RVAs. However, the schema of U-Rel does not model RVAs as columns

with $(rvar \mapsto rval)_*$ in order to restore aliveness. We redefine our definition of aliveness for rows in order to cope with (v_*, d_*). For any *U-Rel*, let r be a row in *U-Rel.rows*:

$$r \hat{\|} getWSD \setminus \{(rvar \mapsto rval)_*\}$$

9.5 Depth

Depth of nodes In order to get a grip on the number of iterations that a glue process has to perform in order to complete, we introduce the depth property. The depth property of a node describes how many viable dependencies a node has:

$$\begin{array}{|l} \text{depth} : \mathit{NODE} \longrightarrow \mathbb{N} \\ \hline \forall\, \mathit{P\text{-}XML} \bullet \\ \quad \forall\, n : \mathit{allnodes} \bullet \text{depth}\, n = \#(\Uparrow_n \cap \mathit{possnodes}) \end{array}$$

Depth of p-documents As we explained in Section 9.3, the G-Join statement associates rows with RVAs such that the WSD of a row describes the world set of the node it represents. Given the fact that (1) we identified a one-to-one correspondence between RVAs and viable dependencies of nodes, and (2) a G-Join adds exactly one RVA to a set of nodes, the number of G-Joins that have to be performed have to be at least as high as the maximal depth value of nodes in a p-document. We define the depth of p-document as the maximum depth of a node in that p-document:

$$\begin{array}{|l} \text{depth} : \mathit{P\text{-}XML} \longrightarrow \mathbb{N} \\ \hline \forall\, pd : \mathit{P\text{-}XML} \bullet \text{depth}\, pd = \max \text{depth}(\!|pd.\mathit{allnodes}|\!) \end{array}$$

It can be proven that for any p-document pd, the optimal number of phases of a glue process is depth(pd).

For illustration purposes, Figure 9.3 presents a partitioning of nodes based on their depth property. It follows that the depth of the p-document in this figure equals 3.

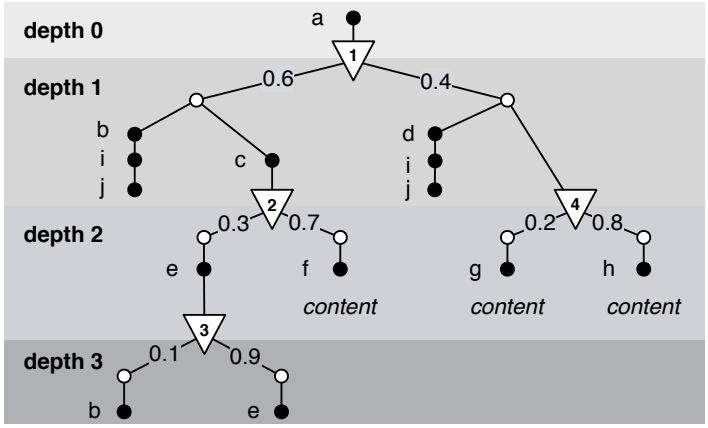


Figure 9.3: Partitioning based on the depth property.

Depth and rdepth of rows In order to get a grip on the progress of a glue process, we define the number of unique RVAs assigned to a row as the depth of that row. Additionally, we define the number of RVAs that have to be assigned to that row as its rdepth.

$$\begin{array}{|l} \text{depth} : \mathit{ROW} \longrightarrow \mathbb{N} \\ \text{rdepth} : \mathit{ROW} \longrightarrow \mathbb{N} \\ \hline \forall\, r : \mathit{rows} \bullet \text{depth}\, r = \#(\mathit{getWSD}\, r \setminus \{(\mathit{rvar} \mapsto \mathit{val})_*\}) \\ \forall\, r : \mathit{rows} \bullet \text{rdepth}\, r = \text{depth}(f^{-1}(r)) \end{array}$$

Function f^{-1} applied to a row returns the node represented by that row.

Depth and rdepth of U-Relations We define the rdepth and depth analogously to the depth of p-documents:

$$\begin{array}{|l} \text{depth} : \mathit{U\text{-}Rel} \longrightarrow \mathbb{N} \\ \text{rdepth} : \mathit{U\text{-}Rel} \longrightarrow \mathbb{N} \\ \hline \forall\, \mathit{rd} : \mathit{U\text{-}Rel} \bullet \text{depth}\, \mathit{rd} = \max \text{depth}(\!|\mathit{rd.rows}|\!) \\ \forall\, \mathit{rd} : \mathit{U\text{-}Rel} \bullet \text{rdepth}\, \mathit{rd} = \max \text{depth}(\!|f^{-1}(\!|\mathit{rd.rows}|\!)|\!) \end{array}$$

A few notes on the depth property of U-Relations. A U-Relation represent RVAs with conditional columns such that each triple of columns represents one RVA for each row. As a consequence, for each U-Relation u, $d \times 3$ conditional columns are used to store the RVAs of u where $d = \text{depth}\, u$. Additionally, we define the WSD of rows as a set of RVAs. In practice, the WSD of rows are implemented as an ordered list of RVAs that may include duplicate RVAs.

Depth of query results The evaluation of SQL queries on U-Relations is similar to the evaluation of patterns on a U-Database. Any row r in the result of any SQL query q derives from one row of each U-Relation specified in the FROM-clause of q. We refer to the rows of which r derives as rs. The WSD of r is created as the WSDs of all rows in rs linked to each other by which duplicates are allowed. As a consequence, the actual depth of r is $\text{depth}(r) = \sum \text{depth}(\!|q.\mathit{from}|\!)$ where $q.\mathit{from}$ contains the set of U-Relations specified in the FROM-clause of q. Since all rows in the result of an SQL query derive from the same tables, the depth of all rows in a query result is the same. Hence, we define the depth an SQL query as:

$$\begin{array}{|l} \text{depth} : \mathit{SQL} \longrightarrow \mathbb{N} \\ \hline \forall\, q : \mathit{SQL} \bullet \text{depth}\, q = \sum \text{depth}(\!|q.\mathit{from}|\!) \end{array}$$

9.6 Summary

This chapter gives an introduction to relational gluing. A relational glue process is constructed of iterations to which we refer as phases. A phase evaluates a G-Join. A G-Join takes a phase table, the skeleton and a glue method in order to return a new phase table that is constructed as the input phase table where each entry is attached with an entry from the skeleton. Skeleton entries and phase table rows are matched by a glue method. This matching process is based on right dependency handles that point to left dependency handles. Right dependency handles are part of the rows of a phase table. Analogously, left dependency handles are part of the entries of the skeleton.

The depth of a row in a phase table specifies how many skeleton entries have to be attached to that row. Not all rows have the same depth. However, all rows have to participate in a relational glue process. In order to ensure that a glue process can attach a skeleton entry to each row of a phase table, the don't care RVA is attached to entries that do not require an extra skeleton entry to be attached to them.

Chapter 10

Glue Processes - The Provision of Uncertainty Awareness

In this chapter, we specify multiple glue processes. We specify a glue process is the combination of one specific glue method with one specific glue method application. A glue process is administered to either the flesh or to the result of a traditional query evaluated on the flesh. In case of the former, a glue process is part of the data mapping process. In case of the latter, a glue process is evaluated as part of the query evaluation process.

This chapter is structured as follows. First, we discuss various flavours of glue method applications in Section 10.1. Second, we introduce three categories of glue methods in Section 10.2. Third, we discuss the administering of a glue process to either the data mapping process or to the query mapping process in Section 10.3.

10.1 Flavours of glue method application

In this section, we discuss three flavours of how a glue method can be applied to the flesh. An application of a glue method can be interpreted as the order in which RVAs are glued to rows. Recall from Section 8.4.2 that we represent the flesh as multiple U-Relations. As a consequence, a glue process has to be applied to each of the U-Relations that represent the flesh. However, in our definitions, we consider the flesh as one U-Relation.

The goal of each glue method application is to transform the flesh fl into a phase table ph_n such that $\text{depth}\, ph_n = \text{rdepth}\, ph_n$. Recall that depth specifies how many RVAs are associated to each row of a U-Relation and that rdepth specifies how many RVAs have to be associated to each row of a U-Relation. An application of a glue method application increases the depth of rows and the depth a phase table but does not affect rdepth.

We define *phases* and *phase tables* per flavour of glue method application. The definition of phases in Section 9.1 corresponds with the batch based application. We tweak this definition for the other flavours of application.

10.1.1 Batch based application

Summary The batch based application (BB) takes a phase table ph_i and adds one RVA to each row. The result is phase table ph_{i+1}. By definition, the following holds: $\text{depth}(ph_{i+1}) = \text{depth}(ph_i) + 1$. If $\text{depth}(ph_{i+1}) < \text{rdepth}(ph_{i+1})$, than BB is applied to ph_{i+1}. This process is repeated until the rdepth and depth of a phase table are equal.

Definition We define phase tables for the BB as follows:

$$
\begin{aligned}
ph_0 &= \mathit{fl} \\
ph_1 &= G_{\bowtie}(ph_0, sk, m) \\
&\vdots \\
ph_{n+1} &= G_{\bowtie}(ph_n, sk, m)
\end{aligned}
$$

The BB application instantiates ph_0 with the flesh. The first phase is defined as the evaluation of $G_{\bowtie}(ph_0, sk, m)$ that transforms ph_0 into ph_1 through some glue method m. Phase table ph_1 is considered to be the result of the first phase. In general, phase $(n+1)$ takes the phase table that is the result of phase n and transforms it into a new phase table that is considered to be the result of phase $(n+1)$. A BB application uses $\text{rdepth}\,(ph_0)$ phases to complete. The last created phase table is the result of the BB application.

Example For illustration purposes, we refer to Figures 10.3 and 10.7. In both figures, the first vertical column presents the flesh of our running example in Figure 2.8. A more detailed presentation of the flesh is found in Figure 9.2b. Each phase appends one RVA to each row of the original flesh. At completion of a glue process that uses BB, each row in ph_0 is appended with rdepth (ph_0) RVAs.

10.1.2 Partition based application

Summary We partition the flesh fl into $m+1$ partitions $\{p_0, \ldots, p_m\}$ where each partition contains all rows with the same rdepth. Since rows have the same rdepth, we know for each partition how many don't care RVAs will be associated to rows. We use this information to optimize the process of assigning don't care RVAs.

Definition Let $\text{rdepth}(ph_0) = rd$. We define phase tables for partition based application (PB) as follows.

$$
\begin{aligned}
ph_0 &= G_{\ltimes}^{(rd,0)}(\{r : \mathit{fl} \mid \text{rdepth}\, r = 0 \bullet r\}, sk, m)\\
ph_1 &= G_{\ltimes}^{(rd,1)}(\{r : \mathit{fl} \mid \text{rdepth}\, r = 1 \bullet r\}, sk, m)\\
&\vdots\\
ph_{n+1} &= G_{\ltimes}^{(rd,n+1)}(\{r : \mathit{fl} \mid \text{rdepth}\, r = n+1 \bullet r\}, sk, m)
\end{aligned}
$$

The PB application applies a partitioning to the flesh $\langle p_0, p_1, \ldots, p_{rd}\rangle = \mathit{fl}$ such that each row r that is part of partition p_i satisfies $\text{rdepth}(r) = i$. Phase table ph_i is constructed as $G_{\ltimes}^{(rd,i)}(p_i, sk, m)$ where rd in (rd, i) denotes the number of G-Joins that are applied to p_i and i in (rd, i) denotes the number of non don't care RVAs that are assigned to each row of p_i. Other G-Joins append don't care RVAs to each row of p_i. The result of the PB application is the union of all generated phase tables. It can be proven that the order in which phase tables are constructed is irrelevant. As advantage, partition based phase tables can be constructed in parallel.

Example For illustration purposes, we refer to Figures 10.3 and 10.7. The horizontal columns in both figures denote PB phase tables. We identify the first vertical column as the flesh that is split into four partitions. For each partition holds that the rows of a partition are associated with the same number of don't care RVAs.

10.1.3 Precomputed chaining

In Section 5.3, we introduce the concept of a skeleton path. The skeleton path $\Uparrow_n$ of a node n represents all viable dependencies of that node as a set of possible choices. In a URDBMS, a viable dependency is represented as an RVA. It follows that the same set of viable dependencies, represented as a skeleton path, can be represented as a set of RVAs.

Summary The idea behind PC is to precompute multiple sets of RVAs in order to represent all skeleton paths in a p-document with a set of U-Relations. We refer to these U-Relations as *skeleton path tables*. A skeleton path table, denotes as $\mathcal{S}_n$, stores all skeleton paths that are constructed of n or less possible choices. Each entry in $\mathcal{S}_n$ corresponds with one skeleton path. Skeleton path tables are computed as part of a data mapping prior to a glue process and after the skeleton mapping. It follows that the use of skeleton path tables require extra storage costs and burden the data mapping with an extra task. The skeleton path tables derived from skeleton 9.2c are illustrated in Figure 10.1.

Definition We define phase tables for PC as follows:

$$ph_0 = fl$$
$$ph_1 = G_{\bowtie}(ph_0, \mathcal{S}_{\text{rdepth}(ph_0)}, m)$$

The PC application instantiates ph_0 with the flesh. Precomputed chaining completes after one single phase. This phase uses one skeleton path table that contains all skeleton paths with a length of at most rdepth(ph_0). We refer to this skeleton path table as $\mathcal{S}_{\text{rdepth}(ph_0)}$. For table ph_0, it holds that each entry r in ph_0 is associated with one entry s in $\mathcal{S}_{\text{rdepth}(ph_0)}$ such that s describes the complete world set of r. Entry s is constructed of multiple RVAs including don't care RVAs. The advantage of this approach is that a glue process does not have to consider gluing with don't care RVAs.

We motivate the design of skeleton path tables with the following. Our flesh mapping ASI[XA] represents the flesh of a p-document as a set of element tables ts. A corresponding glue process has to be applied to each each element table in ts. Since the rdepth property of different element tables may differ, the PC application requires multiple skeleton path tables in order to perform a glue process with an element table t in ts and skeleton path table $\mathcal{S}_{\text{rdepth}}(t)$[1].

Example For illustration purposes, we refer to Figures 10.3 and 10.7. In both figures, the three most right columns merged together present entries in $\mathcal{S}_3$ –skeleton path table $\mathcal{S}_3$ is illustrated in Figure 10.1. The order of RVA entries may differ. It can be proven that whatever the order of RVAs assigned to a row, the same world set is described. Analogously, it can be proven that a skeleton path is represented as an *unordered* set of possible choices.

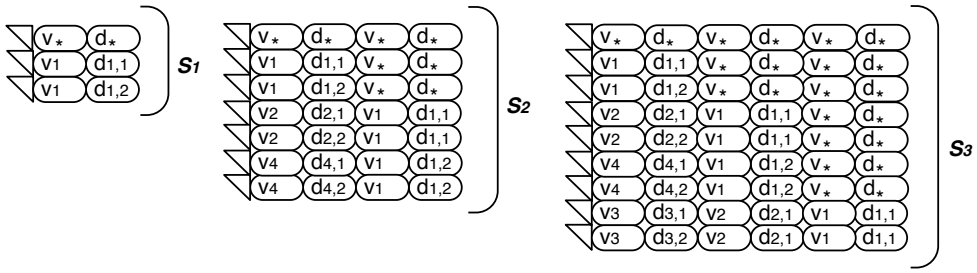


Figure 10.1: Skeleton Path Tables $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$

Precomputed chaining replaces the skeleton with skeleton path tables. A comparison between the skeleton path tables in Figure 10.1 and the skeleton in Figure 9.2c shows that skeleton path tables require extra storage. However, the skeleton path tables do not store right dependency handles.

10.2 Categories

In this section, we introduce various categories of *glue methods*. Glue methods are designed to associate RVAs to rows with solely the right and left dependency handles.

We distinguish two categories of glue methods: flesh driven glue methods –discussed in Section 10.2.1– and skeleton driven glue methods –discussed in 10.2.2. Flesh driven glue methods take the perspective of an ordinary node that is searching for all its ancestor possibility nodes in order to collect all its viable dependencies. In contrast, skeleton driven glue methods take the perspective

[1] If the flesh is represented as one table, only one skeleton path table has to be created.

of a possibility node that associates all its descendants with the viable dependency that captures that possibility node to be selected.

Glue methods operate on a set of rows that represent the flesh of a p-document. The flesh of a p-document is defined as the p-document stripped of all its distributional nodes. Since no distributional nodes reside in the flesh, a skeleton driven glue method should not search for ordinary nodes in the descendant-or-self axis of possibility nodes, but in the descendant axis. Likewise, a flesh driven glue method should search for possibility node in the ancestor axis of ordinary nodes.

10.2.1 Flesh driven glue methods

Possibility parent Flesh driven glue methods take the perspective of an ordinary node that is searching for all its possibility nodes that reside in its ancestor. This search process is accomplished with the concept of a possibility parent. We define the *possibility parent* of a node n to be the possibility node that is reached with the smallest amount of parent location steps. We write $d_{par}(n)$ to refer to the possibility parent of n. If n is represented as a row r, we write $d_{par}(r)$ to refer to the row that represents the possibility parent of n.

Ancestor possibility nodes In order to retrieve all possibility nodes that reside in the ancestor axis of n, we recursively evaluate the possibility parent function. This boils down to:

$$\{d_{par}(n), d^2_{par}(n), \ldots, d^{\mathrm{depth}\,n}_{par}(n)\} = \Uparrow_n \backslash probnodes$$

By definition, the depth property of n is the number of possibility nodes in the ancestors axis of n. Hence, after depth(n) recursive calls of the d_{par}-function, we find all possibility nodes in the ancestor axis of n.

Example An illustration of the d_{par}-function is found in Figure 10.2. We visualize $\uparrow_n$, the path from n to the root of a p-document. Path $\uparrow_n$ defines the ancestor-or-self axis of n. In order to find all possibility nodes that reside in the ancestor axis of n, we apply d_{par} to n various times.

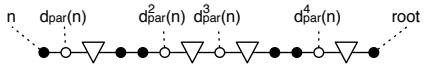


Figure 10.2: Identification of possibility nodes in the ancestor axis of a node n

Goal: preserve aliveness by adding all viable dependencies According to Theorem 1, a node n is part of those possible worlds that select all possibility nodes in the ancestor axis of n. Function d_{par} is able to find all ancestor possibility nodes of a particular node. We rewrite Theorem 1 as a statement that uses d_{par}:

$$n \hat{\|} \{\triangledown\, d_{par}(n), \triangledown\, d^2_{par}(n), \ldots, \triangledown\, d^{\mathrm{depth}\,n}_{par}(n)\}$$

Function $\triangledown$ –defined in Section 5.1– applied to a possibility node returns the possible choice of that possibility node to be selected. More specifically: $\triangledown d = (v \mapsto d)$ for a possibility node d and its parent probability node v.

For P-XML into URDBMS data mapping f, our goal is to construct a glue process that uses a flesh driven glue method in order to preserve aliveness under f:

$$f\ n \hat{\|} f(\{\triangledown\, d_{par}(n), \triangledown\, d^2_{par}(n), \ldots, \triangledown\, d^{\mathrm{depth}\,n}_{par}(n)\}) \iff$$
$$f\ n \hat{\|} \{\triangledown\, d_{par}(f\ n), \triangledown\, d^2_{par}(f\ n), \ldots, \triangledown\, d^{\mathrm{depth}\,n}_{par}(f\ n)\})$$

We identify $f\ n$ as a row that represents a node in the flesh of a p-document. Analogously, we identify $\triangledown\, d^i_{par}(f\ n)$ as an RVA that represents a possible choice in the skeleton of a p-document.

Approach: glue viable dependencies In order to preserve aliveness, we have to restore viable dependencies. From a high level perspective, we accomplish this as follows. For a glue process that uses a BB application and a flesh driven glue method m, we design m such that:

$$
\begin{aligned}
&ph_0 = \mathit{fl}\\
&ph_1 = G_{\ltimes}(ph_0, sk, m) \text{ such that } \textstyle\bigwedge_{i=1}^{1} \forall\, r : ph_1 \bullet r \parallel \triangledown d^i_{par}(r)\\
&\vdots\\
&ph_{n+1} = G_{\ltimes}(ph_n, sk, m) \text{ such that } \textstyle\bigwedge_{i=1}^{n+1} \forall\, r : ph_{n+1} \bullet r \parallel \triangledown d^i_{par}(r)
\end{aligned}
$$

In the first phase, each entry r in ph_0 that represents a node n is made viable dependent on the RVA $\triangledown d^1_{par}(r)$ that represents the possible choice of the possibility parent of n to be selected. The G-Join statement ensures that all rows in ph_0 are also present in ph_1. In the second phase, the same process is repeated for the possibility grandparent of n, denoted as RVA $\triangledown d^2_{par}(r)$. Each following phase ensures that r is made viable dependent on a possible choice that resides one depth higher in the p-document. It can be proven that the number of phases required to complete the full glue method is equal to $\mathrm{rdepth}\, ph_0$. Any phase executed after a glue process is completed will only add don't care RVAs to each row of the input phase table.

Example For illustration purposes, we show the result of any flesh driven glue method applied to our running example in Figure 2.8. We obtain Figure 10.3 that is derived from the flesh and skeleton in Figures 9.2b and 9.2c. Phase 0 denotes the flesh that is in more detail shown in Figure 9.2b. A closer look at phase 1 shows that each row r in ph_0 that represents a node n is associated with an RVA of the form $(v \mapsto d)$ such that $d_{par}\, n = d$. The only exception is the first row that denotes the don't care RVA. This follows from the fact that the first entry of the flesh represents a node that has no possibility parent. Likewise, phase 2 assigns an RVA to rows in ph_0 that involve the possibility grand parent of each row.

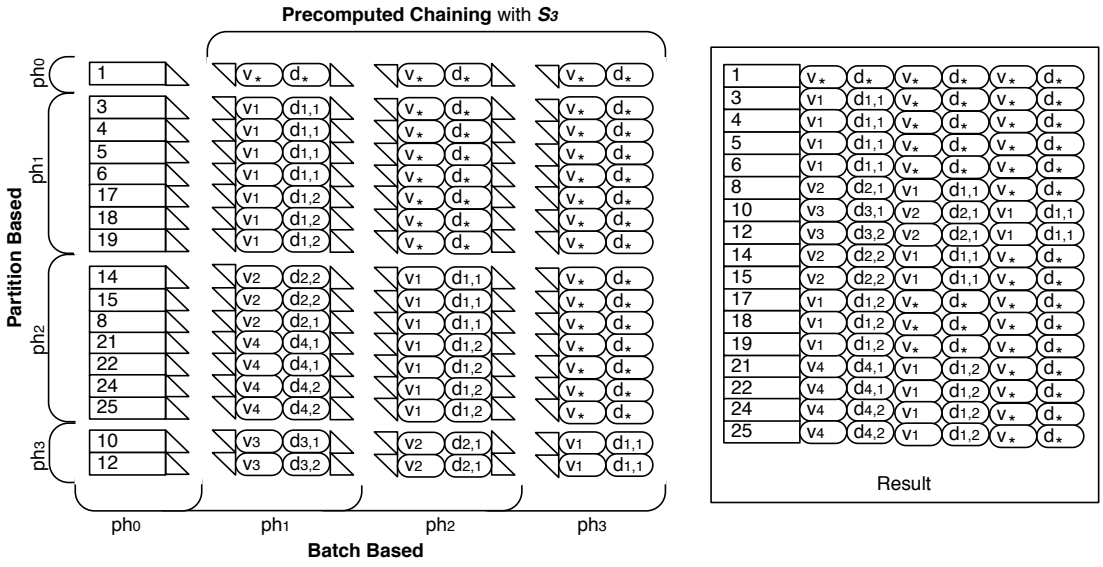


Figure 10.3: The result of a glue process that uses a flesh driven glue methods

Next, we define three flesh driven glue methods. A glue method does nothing more than associating entries in the flesh with entries in the skeleton using their dependency handles. Per glue method, we first define the format of dependency handles and then define how a glue method relates right dependency handles to left dependency handles.

10.2.1.1 Glue by possibility parent reference

From a high level perspective, we design glue by possibility parent reference (PPR) as a pointer from the flesh to the skeleton.

Design for dependency handles We design dependency handles for PPR as follows.

1. For each entry s in the skeleton that represents possible choice $(v \mapsto d)$, we define $pre(d)$ to be the left dependency handle of s. Pre-order numbering is discussed in Section 2.3.2. We introduce the column *pre* to hold the left dependency handles for the skeleton.

2. For each entry s in the skeleton that represents possible choice $(v \mapsto d)$, we define $pre(d_{par}(d))$ to be the right dependency handle of s. We introduce the column *posspre* to hold the right dependency handles for the skeleton.

3. For each entry r in the flesh that represents node n, we define $pre(d_{par}(n))$ to be its right dependency handle. We introduce the column *posspre* to hold the left dependency handles for the skeleton.

Nodes that do not have a possibility parent use the pre-order of the don't care point as their right dependency handle. This includes the don't care RVA as well. As a consequence, the don't care RVA has a right dependency handle that points to itself.

Definition for phases We define a phase for PPR as follows. Let ph_i be a phase table and sk be the skeleton. In phase i, a G-Join $ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{ppr})$ glues ph_i and sk into a phase table ph_{i+1} as follows:

$$ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{ppr}) = \{r : ph_i, s : sk \mid r.posspre = s.pre \bullet (r.pre, r.size, s.posspre)\}$$

Glue method m_{ppr} is represented as join condition $(ph_i.posspre = sk.pre)$ of this G-Join. Row attributes $(r.pre, r.size)$ represent a certain node n in the flesh. Row-attribute $r.posspre$ denotes the right dependency handle of r for phase i. Row s represents the possible choice of the possibility parent of node $f^{-1}(r)$ to be selected. Observe that we assign the right dependency handle of s to r. As a consequence, in phase $i+1$, r is associated with row s' such that $f^{-1}(s') = d_{par}^{i+1}\, n$. The transfer of the right dependency handle of the skeleton to the flesh allows for the recursive evaluation of the possibility parent reference such that the transitive closure of the possibility parent function can be evaluated.

Example For illustration purposes, Figure 10.4 shows two phases of the glue process BB.PPR for three entries in the flesh. This figure corresponds with Figure 10.3. We observe a table that represents the flesh and a table that represents the skeleton. In each BB phase, a skeleton table is used. Since we show two phases, we also show two skeleton tables. In the first phase, the right dependency handles in column *fl.posspre* are matched with the left dependency handles in *sk.pre*. The glue method PPR uses these two columns in order to append one skeleton entry to each flesh entry. Afterwards, columns *fl.posspre* and *sk.pre* are discarded. In the second phase, this process is repeated. Only this time, *sk.posspre* holds right dependency handles. Since all dependency handles are discarded, the result of BB.PPR contains each entry in the flesh appended with RVAs from the skeleton. The full result of BB.PPR is illustrated in Figure 10.3.

10.2.1.2 Glue by closest dependency

From a high level perspective, we design glue by closest dependency (CD) as an aggregate that associates entries in the flesh with the closest ancestor choice in the skeleton.

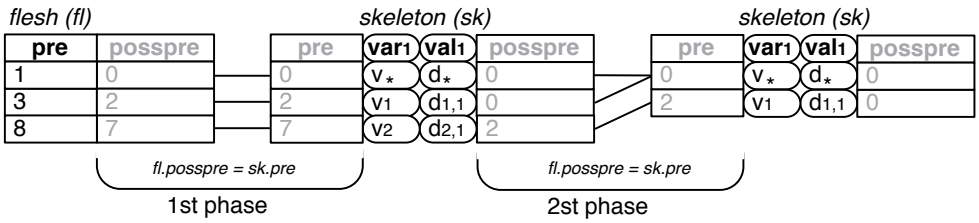


Figure 10.4: The glue process BB.PPR for three entries in the flesh.

Design of dependency handles We design dependency handles for CD as follows.

1. For each entry s in the skeleton that represents possible choice $(v \mapsto d)$, we define $(pre(d), size(d))$ to be the left dependency handle of s. Pre-order numbering and the size-property are discussed in Section 2.3.2. We introduce the columns *pre* and *size* to hold the left dependency handles for the skeleton.
2. For each entry s in the skeleton that represents possible choice $(v \mapsto d)$, we define $pre(d)$ to be the right dependency handle of s. Column *pre*, introduced as part of the left dependency handle, already stores pre-order.
3. For each entry r in the flesh that represents n, we define $pre(n)$ to be its right dependency handle. The pre-order numbering is part of the element encoding. We consider column *pre* to hold the right dependency handles of the flesh. Since the pre-order is part of the element encoding, we consider column pre' to hold a copy.

In contrast to PPR, we do not have a mechanism that enables the right dependency handle of the don't care RVA to point to itself. We deal with this problem in Section 11.3. This problem is also illustrated in the example at the bottom of this subsection.

Definition of phases We define a phase for CD as follows. Let ph_i be a phase table and sk be the skeleton. In phase i, a G-Join $ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{cd})$ glues ph_i and sk into a phase table ph_{i+1} as follows:

$$\begin{aligned} ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{cd}) = \{ & r : ph_i, s : sk \mid \\ & s.pre = \max\{s' : sk \mid s'.pre < r.pre \leq s'.pre + s'.size \bullet s'.pre\} \\ & \bullet (r.pre, r.size, s.pre) \\ \} & \end{aligned}$$

Glue method m_{cd} represents the join condition of this G-Join. This condition is derived from the proof of Appendix A.1; for any node n, its possibility parent corresponds with its closest possibility node. Analogously to PPR, we transfer the right dependency handles of the skeleton to the flesh. This allows for the recursive execution of the possibility parent reference such that we can evaluate the transitive closure of the possibility parent function.

Example For illustration purposes, Figure 10.5 shows two phases of the glue process BB.CD for three entries in the flesh. We illustrate the same matching as in previous examples. Note that one of the matches is denoted with a dotted line. This match does not satisfy the join condition specified by m_{cd}. We solve this problem in Section 11.3. The full result of BB.CD for the complete skeleton is illustrated in Figure 10.3.

10.2.1.3 Glue by sandwich

From a high level perspective, we design glue by sandwich (SW) as a complex condition that also associates entries in the flesh with the closest ancestor choice in the skeleton. We design dependency handles for SW analogously to the design of dependency handles for CD.

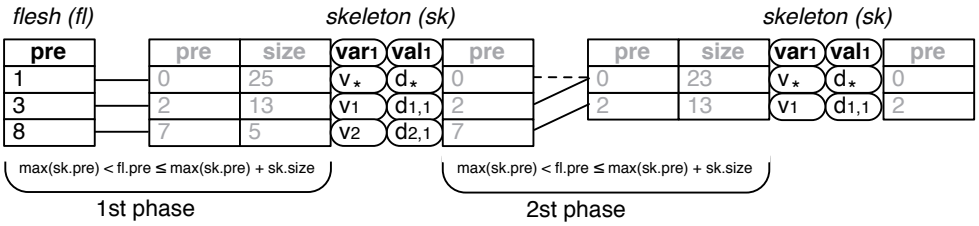


Figure 10.5: The glue process BB.CD for three entries in the flesh.

Definition of phases We define a phase for SW as follows. Let ph_i be a phase table and sk be the skeleton. In phase i, a G-Join $ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{sw})$ glues ph_i and sk to a phase table ph_{i+1} as follows:

$$
\begin{aligned}
ph_{i+1} = G_{\bowtie}(ph_i, sk, m_{sw}) = \{ & r : ph_i, s : sk \mid \\
& s.pre < r.pre \leq s.pre + s.size \wedge \\
& s.pre \notin \{ s' : sk \mid \\
& \qquad s'.pre < r.pre \leq s'.pre + s'.size \wedge \\
& \qquad s.pre < s'.pre \leq s.pre + s.size \\
& \qquad \bullet\ s'.pre \} \\
& \bullet\ (r.pre, r.size, s.pre) \\
\}
\end{aligned}
$$

Row r represents node n. Row s represents the possibility parent n if there does not exist a row s' that represents a possibility node that is (1) an ancestor of n and (2) a descendant of s. If no such s' exists, node s has to be the closest possibility node of n. Hence, s represents the possibility parent of n. The condition that defines s is m_{sw}. Analogously to PPR, we transfer the right dependency handles of the skeleton to the flesh. This allows for the recursive execution of the possibility parent reference such that we can evaluate the transitive closure of the possibility parent function.

Example For illustration purposes, we explain SW on the basis of Figure 10.6. On top of this figure, the path of n to the root node is illustrated. We observe that n has four ancestor possibility nodes. In order to determine which of these four ancestor possibility nodes is the possibility parent of n, each path of n to one of its ancestor possibility nodes is processed by SW. For path $\uparrow_{n,d_i}$, glue method SW establishes that no possibility node lies between n and d_i. This in contrast to all other paths. It follows that d_i is $d_{par}(n)$.

10.2.2 Skeleton driven glue methods

Possibility node descendants Skeleton driven glue methods take the perspective of a possibility node that is searching for all ordinary nodes in the descendant axis such that each descendant is associated with the choice that selects its ancestor possibility node.

Simultaneous search for descendants Two possibility nodes are allowed to search simultaneously for their descendants if they have no descendants in common. In this case, we say that possibility nodes are *independent* of each other. This constraint follows from the design of the G-Join statement. We optimize the number of independent possibility nodes by exploiting their depth property. We defined the depth property as the number of possibility nodes that reside in the ancestor-or-self axis of a node. The depth property of possibility nodes can also be defined as their level in the skeleton, excluding probability nodes. It can be proven that if two possibility nodes have the same depth, they relate to each other on the preceding/following axis. Grust et

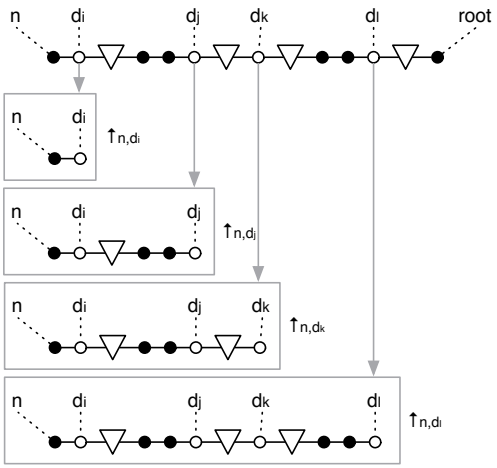


Figure 10.6: Illustration of the paths from n to each of its ancestor possibility nodes

al. [20] observe that nodes in a tree-structure that relate to each other on the preceding/following axis have no descendants in common. We conclude: possibility nodes that have the same depth are independent of each other. Hence, we design skeleton driven glue methods to perform a G-Join with all the possibility nodes that have the same depth value.

Restore viable dependencies Skeleton driven glue methods are designed to associate skeleton-entries to flesh-entries per depth level. A glue process that applies a skeleton driven glue method with a BB application associates RVAs as follows:

$$
\begin{aligned}
&ph_0 = \mathit{fl} \\
&ph_1 = G_{\bowtie}(ph_0, sk, m) \text{ such that } \forall\, r : ph_1 \bullet r \parallel ds \text{ where} \\
&\qquad ds = \{(v \mapsto d) : sk \mid f^{-1}(r) \in d/\texttt{descendant} \wedge\ d.depth \leq 1 \bullet (v \mapsto d)\} \\
&\ \vdots \\
&ph_{n+1} = G_{\bowtie}(ph_n, sk, m) \text{ such that } \forall\, r : ph_{n+1} \bullet r \parallel ds \text{ where} \\
&\qquad ds = \{(v \mapsto d) : sk \mid f^{-1}(r) \in d/\texttt{descendant} \wedge\ d.depth \leq (n+1) \bullet (v \mapsto d)\}
\end{aligned}
$$

In the first phase, for each entry r in ph_0 that represents a node n under f, r is associated with the RVA that represents the possible choice of the possibility node with a depth of 1 in the ancestor axis of n to be selected. In the second phase, the same process is repeated for the possible choice that selects the possibility node with a depth of 2 in the ancestor axis of n. If a node n does not have an ancestor possibility node at a specific depth, the don't care RVA is assigned to r.

Example For illustration purposes, we show the result of a skeleton driven glue method applied to our running example of the p-document in Figure 9.3. We obtain Figure 10.7 that is derived from the flesh and skeleton in Figures 9.2b and 9.2c. Phase 0 denotes the flesh. In the first BB phase, nodes in the flesh with a depth value larger or equal to one are associated to their ancestor possibility node with a depth of one. As illustrated in Figure 9.3, $d_{1,1}$ and $d_{1,2}$ are the only two possibility nodes that satisfy this criteria. Furthermore, we observe that the root node is the only node that is not a descendant of one of these two possibility nodes. As a consequence, the first row in the the flesh is associated with the don't care RVA.

Most optimal It can be proven that a glue process that applies our defined skeleton driven glue method completes with the most optimal number of phases. Proof sketch: for any glue process, the number of phases is optimal if there exists an entry in the result that is not associated with a

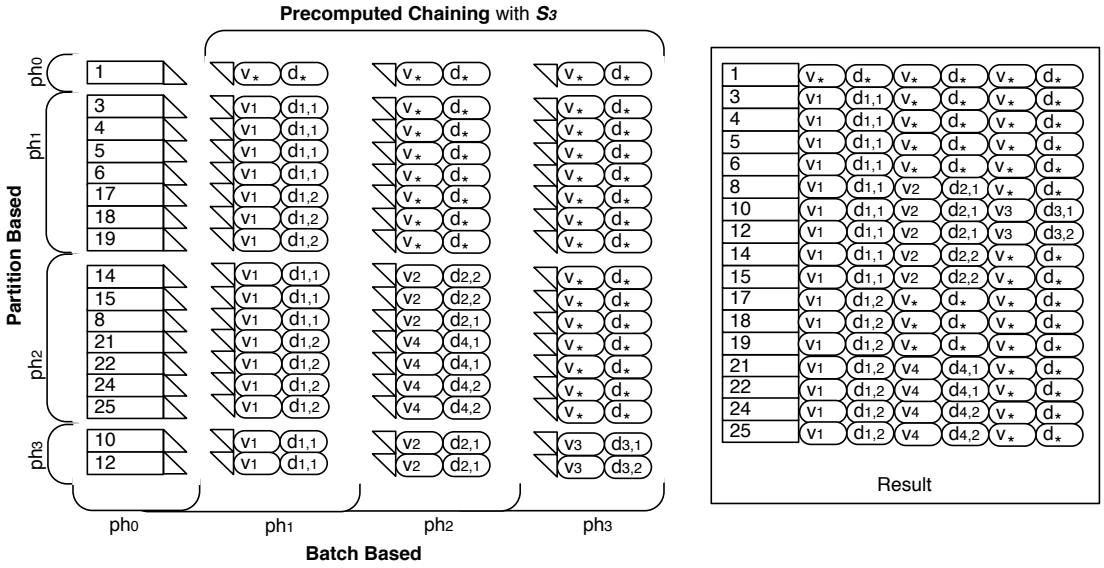


Figure 10.7: The result of a glue process that uses a skeleton driven glue methods

don't care RVA. Otherwise, a more optimal result could be obtained by eliminating one don't care RVA for each entry in the result of a glue process. Such result would be equivalent to the former since don't care RVAs do not change the uncertainty distribution. According to the definition of $(\text{rdepth}\, ph_0)$, there exists a node n in the p-document with a depth equal to $(\text{rdepth}\, ph_0)$. As a consequence, for each depth value in $\{1, 2, \ldots, (\text{rdepth}\, ph_0)\}$, n has a possibility ancestor for each of those depth values. It follows from the definition of phase tables that entry r that represents n cannot be associated with a don't care RVA for the first $(\text{rdepth}\, ph_0)$ phases. We conclude that the number of phases for the skeleton driven glue method is optimal. □

10.2.2.1 Glue by depth

From a high level perspective, we design glue by depth (D) as an expression that requests for the ancestor possibility node of a specific depth.

Design of dependency handles We design dependency handles for the D as follows.

1. For each entry s that represents possible choice $(v \mapsto d)$, we define $(pre(d), size(d), depth(d))$ to be the left dependency handle of s. Pre-order rank and size are discussed in Section 2.3.2. The depth property is discussed in Section 9.5. We introduce the columns *pre*, *size* and *depth* to hold the left dependency handles for the skeleton.

2. For each entry r in the flesh that represents n, we define $pre(n)$ to be the right dependency handle of r. We consider column *pre* to hold the right dependency handles of the flesh. Since the pre-order is part of the element encoding, we consider column *pre* to be already present. In case D is applied with the PC application, the right dependency handles of the flesh also include *depth* n as right dependency handle.

3. Glue method D does not define right dependency handles for the skeleton.

Definition of phases We define a phase for the D glue method as follows. Let ph be a phase table, d be a specific depth value and sk be the skeleton. A G-Join $ph_{n+1} = G_{\bowtie}(ph_n, sk, m_d)$ glues

ph_n and sk to a phase table ph_{n+1} as follows:

$$ph_{i+1} = G_{\bowtie}(ph_i, sk, m_d) = \{r : ph_i, s : sk \mid \\ depth(s) = i \wedge pre(s) < pre(r) \leq pre(s) + size(s) \bullet (pre(r), size(r))\}$$

We require each phase table that is created with D to use $d \in \{1, \ldots, \text{rdepth}\, ph_0\}$ that is unique per phase. For convenience, we use i. Entries that do not have an ancestor possibility node at a specific depth are associated with the don't care RVA.

Example For illustration purposes, Figure 10.5 shows two phases of the glue process BB.D for three entries in the flesh. This example matches the same skeleton entries with flesh entries as in the flesh driven glue method examples. Since glue method D does not define right dependency handles for the skeleton, we prefer to put the flesh in middle. We observe that the matching conditions use a fixed depth value. For a depth value of d, possibility nodes at depth d are matched. In the first phase, possibility nodes with a depth of 1 are matched. The first flesh entry does not have a possibility node with a depth of 1. Hence, this entry is matched with the don't care RVA. In the second phase, possibility nodes with a depth of 2 are matched. Only the last entry of the flesh has a possibility node ancestor with a depth of 2. The other two flesh entries are matched with the don't care RVA. The full result of BB.D for the complete skeleton is illustrated in Figure 10.7.

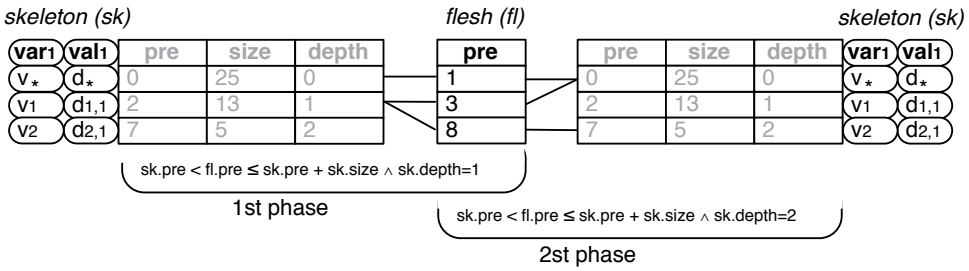


Figure 10.8: The glue process BB.D for three entries in the flesh.

10.2.3 Inheritance driven glue methods

Principle of inheritance Inheritance driven glue methods take the perspective of an ordinary node that is searching for its ordinary node ancestors such that it can copy or inherit their viable dependencies. Prior to this search process, direct children of possibility nodes are associated with the viable dependency that captures their possibility parent to be selected. Otherwise, there would be no viable dependencies to inherit.

First phase In the first phase of an inheritance driven glue method, each flesh entry r that represents node n is associated with an RVA that represents the choice of the direct parent of n to be selected. We consider two scenarios. In the first scenario, d is a possibility node and the direct parent of n. In this case, we associate r with the RVA that selects d. In the second scenario, n' is an ordinary node and the direct parent of n. In this case, we associate r with the don't care RVA.

Following phases In the second phase, each flesh entry $r \in \mathit{fl}^+$ that represents node n is associated with its closest ordinary node n' represented as $r' \in \mathit{fl}^+$. If n' does not exist, r is associated with the don't care RVA. If n' does exist, r inherits the RVA that is associated with r'. It follows from the first phase that this RVA is either the don't care RVA or the RVA that selects the direct parent of n'. We refer to the result of this phase as ph_2.

In order to obtain ph_{n+1}, we perform the same approach as to obtain ph_2 but we take r from ph_n and r' from ph_1. An inheritance glue method is finished if the node at the deepest level in the flesh has visited all its ordinary node ancestors.

Example For illustration purposes, Figure 10.9 shows the result of an inheritance driven glue method applied to our running example in Figure 2.8. Phase 0 denotes the flesh. Each row in the flesh is associated with (1) three gray left dependency handles, (2) three gray right dependency handles, and (3) one white right dependency handle.

In the first phase, white dependency handles are used to associate row r that represent node n the RVA that represents the possible choice that selects the direct possibility parent of n. If n does not have a direct possibility parent, r is associated with the don't care RVA.

In the second phase, each row r in ph_1 is associated to another row r' in ph_1' –a copy of ph_1– such the gray right dependency handle of r matches the gray left dependency handle of r'. If the right dependency handle of r matches the left dependency handle of r', r is child of r'. The result of the second phase is phase table ph_2 for which holds that each row $r \in ph_1$ is also a row in ph_2 and the new world set descriptor of r is equal to the old world set descriptor of r appended with the RVA of r'. The third and fourth phase also use ph_1' to find ordinary node parents.

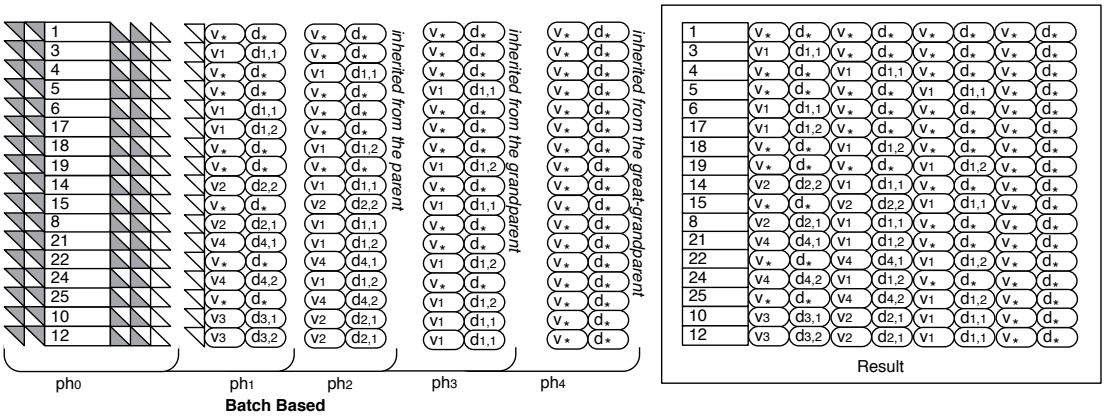


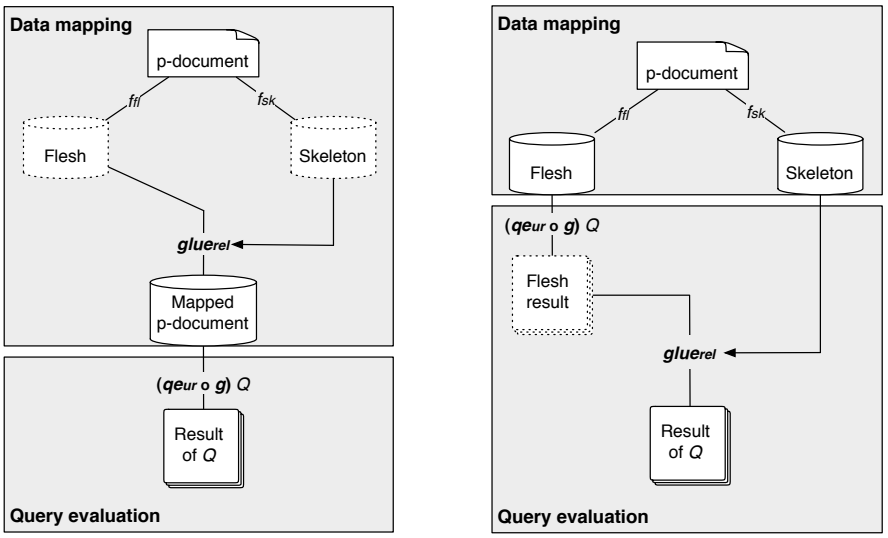
Figure 10.9: The operation of Inheritance Driven Glue Methods

Unsuitable in combination with ASI[XA] Our design of a P-XML into URDBMS mapping is based on ASI[XA]. This XML into RDBMS mapping scatters nodes over different element tables by which it is hard to evaluate the parent axis for all entries in the flesh. However, inheritance driven glue methods could be suitable for other mappings where parent nodes are not scattered. For example, XA that represents a complete tree in one relational table.

Additionally, inheritance driven glue methods are not as efficient as the other two glue method categories. In our previous example, the last phase appends to each row the don't care RVA. In more general, the maximum level in the flesh determines the number of required phases of an inheritance driven glue method. In contrast, the maximum depth in the skeleton determines the number of required phases of a flesh driven or skeleton driven glue method.

10.3 Glue administering

A glue process associates world set descriptors to contents derived from a p-document. Such process has to be performed prior to query evaluation or during query evaluation in order to ensure query evaluation adheres to the possible worlds semantics. We refer to the former as document oriented gluing (DO) and to the latter as query result oriented gluing (QRO). DO glue processes are administered to the flesh. As a consequence, a DO glue process is performed as part of the data mapping. We discuss DO glue processes in Section 10.3.1. QRO glue processes are administered the result of a query evaluated on the flesh. As a consequence, a QRO glue process is performed for each query as part of its query evaluation. We discuss QRO gluing in Section 10.3.2.



(a) Document oriented gluing (b) Query result oriented gluing

Figure 10.10: Glue administration alternatives

10.3.1 Document oriented gluing, Administering a glue process to the data mapping process

We define document oriented gluing (DO) as the administering of a glue process to the contents of a document. As a consequence, a DO glue process is evaluated as part of the data mapping. This process is illustrated in Figure 10.10a. We map flesh and skeleton into a URDBMS. A relational glue process returns a set of U-Relations that represents the same set of possible worlds as the original p-document. Flesh and skeleton should be interpreted as temporary results. They are discarded after a DO glue process has finished.

The uncertainty management mechanism of a URDBMS supports the evaluation of traditional queries with qe_{ur} on U-Relations. We use a traditional XPath to SQL mapping g to evaluate XPath query Q on the result of a DO glue process. We refer to the result of g applied to XPath expressions as *t-queries*.

Altering of the inlining principle DO gluing affects the inlining principle of the ASI[XA] mapping –discussed in Section 8.4.2. The traditional inline rules –described in Section 2.3.1– do not take in account that a set of nodes represented as a single row have to be part of the same set of possible worlds. However, in U-Rel, WSDs describe the world set of single rows. In order to represent multiple nodes in one single row, they have to be part of the same world set. We capture this requirement with a third inline rule: for a host element kind N and guest element kind N', we require that each guest node $n' \in N'$ is part of the same set of possible worlds as the host node $n \in N$ it visits. As a consequence, for each set of nodes represented as one row, they are part of the same set of possible worlds. It follows that one world set descriptor suffices to describe this set of possible worlds.

For illustration purposes, we apply the $ASI[XA]$ approach with the new inline rules to our running example in Figure 2.8. The result is found in Figure 10.11. A comparison with Figure 8.3 shows that 5 extra element tables are created as a result of the new inline rule.

columns	row
a_pre	1
a_size	24
a_par	-1
eT of *a*, rd=0	

columns	row
f_pre	14
f_par	1
f_pcdata	*content*
eT of *f*, rd=2	

columns	row	row
b_pre	3	10
b_size	2	0
b_par	1	8
eT of *b*, rd=3		

columns	row
c_pre	6
c_size	9
c_par	1
eT of *c*, rd=1	

columns	row
g_pre	21
g_par	1
g_pcdata	*content*
eT of *g*, rd = 2	

columns	row	row
i_pre	4	18
i_par	3	17
j_pre	5	19
eT of *i*, rd=1		

columns	row
d_pre	17
d_size	2
d_par	1
eT of *d*, rd=1	

columns	row
h_pre	24
h_par	1
h_pcdata	*content*
eT of *h*, rd = 2	

columns	row	row
e_pre	8	12
e_size	4	0
e_par	6	8
eT of *e*, rd=3		

Figure 10.11: $ASI[XA]$ approach applied to Figure 2.8 with new inline rules

10.3.2 Query result oriented gluing, Administering a glue process to the query mapping process

We define query result oriented gluing (QRO) as an administering of a glue process to query results such that a glue process itself is evaluated as part of the query evaluation process. Figure 10.10b gives an illustration of a QRO glue process. Flesh and skeleton are extracted from the p-document and mapped into a URDBMS. However, they are not glued together afterwards. Instead, we store the mapped flesh and skeleton separately.

Traditional XPath queries are mapped with a query mapping g and executed on the mapped flesh. We define the result of a query evaluated on the flesh as the *flesh result* of that query. Note that the flesh of a p-document represents all the content of that p-document. The only information missing is in which possible worlds what content resides. Consequently, all content is assumed to be in one world. Since this assumption is incorrect, a query evaluated on this one world will return too many results. However, we are certain that the result of a query evaluated on the mapped p-document is contained by its flesh result. In order to retrieve the correct result, we have to describe the world set of each entry in the flesh result. We accomplish this with a QRO glue process applied to the flesh result. SQL queries derived from an XPath query that include a QRO glue process are referred to as tg-queries. This kind of queries operate directly on flesh and skeleton.

Example For illustration purposes, we illustrate the query result oriented glue process in Figure 10.12. In the left upper corner, we present pattern Q that request for all occurrences of a node of element kind e that is preceded by a node of element kind b. We wish to evaluate Q on our running example in Figure 2.8 as a tg-query on flesh and skeleton. The flesh of Figure 2.8 is illustrated in Figure 10.11 as a set of element tables. The first step is to evaluate Q on the flesh. Since Q only request for nodes of element type b and element type e, we perform Q on element tables eT_b and eT_e. The result is illustrated in the right upper corner. We refer to the result of Q evaluated on the flesh as the flesh result of Q. The flesh result of Q is a set of three rows where each row is constructed of 6 row-attributes and two right dependency handles, one per used element table. Next, we perform a relational glue process on the flesh result of Q that replaces dependency handles with RVAs. More specifically, dependency handles derived from eT_b are replaced with 3 RVAs since $\text{rdepth}(eT_b) = 3$ and dependency handles derived from eT_e are replaced with 2 RVAs since $\text{rdepth}(eT_e) = 2$. We obtain the result of Q in the bottom. Observe that the third row is associated with conflicting RVAs; values $d_{3,1}$ and $d_{3,2}$ are assigned to v_3. Hence, this row is deleted.

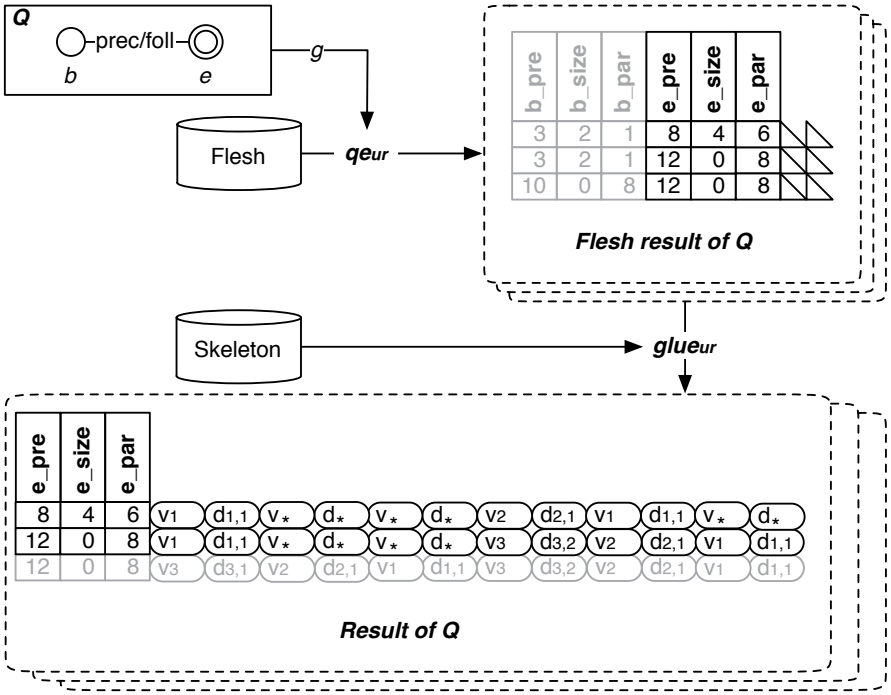


Figure 10.12: Query result oriented gluing

Application of the inlining principle Rows in the flesh are not associated with a world set descriptor. As a consequence, multiple nodes that are represented as one row do not have to be part of the same world set. This allows us to use the traditional set of inline rules for the flesh mapping. However, if multiple nodes are represented as one row, we require that row to be associated with a right dependency handle for each node it represents. As a consequence, element tables can become very large to store all extra dependency handles. However, for the glue methods SW and CD, the right dependency handle of element tables is part of the element encoding and comes free of storage costs. We consider a flesh mapping that uses the traditional set of inline rules to be suitable for these two glue methods.

10.3.3 Qualitative comparison between document oriented and query result oriented gluing

A glue process as part of the evaluation of each query will slow query performance. The avoidance of a glue process as part of a data mapping will improve performance. A lot of duplicate RVAs are stored if a data mapping includes a glue process. Therefore, it is more efficient to store skeleton and flesh separately. If the skeleton and flesh are stored separately, a loosening of the inline rules can be applied. As a consequence, more element kinds can be stored together which reduces storage costs and improves the evaluation of the parent/child axis. *tg*-queries have the advantage that a URDBMS can decide if it is more efficient to first perform a glue process or the flesh result. *tg*-queries have the advantage that they make uncertainty management in a URDBMS tree-aware, by which query evaluation is made more efficient.

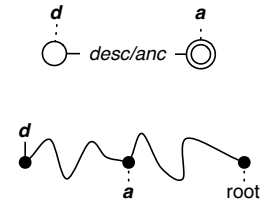
10.4 Tree-aware uncertainty management

The U-Relational data model does not take in account that some rows have RVAs in common. As a consequence, duplicate RVAs may be assigned to rows in a query result. In this section,

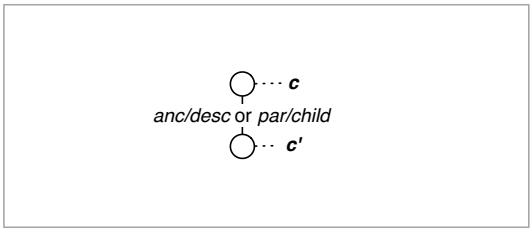
we propose to reuse the tree-structure of an XPath expression to identify what expressions add duplicate RVAs to query results.

In p-documents, the world set of a node n is described as a set of possible choices represented as $\Uparrow_n$, the skeleton path of n. We write $\Uparrow_{n_a} \subseteq \Uparrow_{n_d}$ to state that a node n_a is viable dependent on the same possible choices as a node n_d. A node n_a is viable dependent on the same possible choices as a node n_d if all nodes that lie on path $\uparrow_{n_a}$ also lie on $\uparrow_{n_d}$. It follows that all nodes that lie on $\uparrow_{n_a}$ are part of the ancestor-or-self axis of n_d, including node n_a.

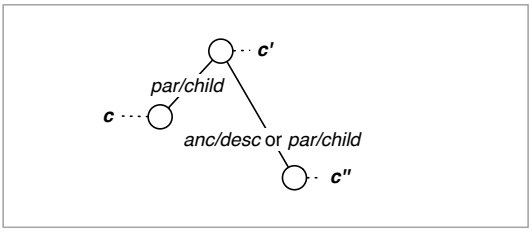
Theorem 2. *Two nodes that share the ancestor/descendant relation have all possible choices of the ancestor node in common.*



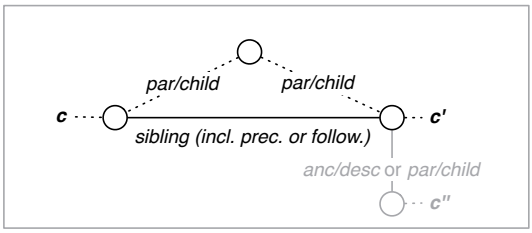
(a) Nodes n_d and n_a instantiate candidates c_d and c_a



(b) Application scenario of 1st tree-awareness rule



(c) Application scenario of the 2nd tree-awareness rule



(d) Application scenario of the 3th tree-awareness rule

Figure 10.13: Application scenario of tree-awareness rules

Next, we apply theorem 2 to three scenarios. Each scenario is represented as a pattern. For each pattern, we argue that an instantiation adds duplicate RVAs to the answer of that pattern. These duplicate RVAs originate from instantiations of a certain candidate c. We use this information to reduce number of RVAs that are appended to the flesh result during a query result oriented glue process. We describe the scenarios on the basis of the the terminology introduced in Chapter 4. We give a short summary of our terminology. A pattern is constructed of candidates. Candidates are related to each other through edges that represent required relations, or in this case, required XPath axes. A combination of nodes instantiate a pattern if they match the structure specified in the pattern. The answer to a pattern is a subset of nodes that instantiate the pattern. This set of nodes is associated with a world set descriptor that is the union of the world set descriptors of all nodes that instantiate the pattern.

1st tree-awareness rule The first scenario is illustrated in Figure 10.13b. We refer to this scenario as twig pattern p_1. We observe that p_1 is constructed as two candidates $-c$ and $c'-$ where c' is a descendant of c. We denote an instantiation of p_1 as $\langle n, n' \rangle$ such that node n instantiates candidate c and node n' instantiates candidate c'. According to Theorem 2, for any two nodes related through the ancestor/descendant axis, the following holds $\Uparrow_{n'} \subseteq \Uparrow_n$. Therefore, for each instantiation $\{n, n'\}$, possible choices of n do only add duplicate possible choices to the result of p_1.

From this observation, we extract the following rule: let $\langle c, c' \rangle$ be two candidates in a certain twig pattern p where c is an ancestor of c'. For any instantiation of c with a node n, no possible choices of n are added to the world set descriptor of an instantiation of p, because these choices are already added by the instantiation of c'. We refer to this rule as *the first tree-awareness rule.*

For illustration purposes, we apply the first tree-awareness rule to the following XPath query: $Q_{P\text{-}XML}$ = `//actor[name/text()='Akhtar, Ayad]/child::role/sibling::role` that is visualized in Figure 10.14. In order to evaluate $Q_{P\text{-}XML}$ with a URDBMS, we map $Q_{P\text{-}XML}$ to its SQL variant Q_{SQL}. We evaluate Q_{SQL} on the element tables of the used element kinds. We present the rdepth of these tables in Figure 10.14. A simple calculation shows that the depth of Q_{SQL} equals 17. This means the each row in the result of Q_{SQL} is appended with 17 RVAs. We reduce this number by 3 with an application of the first tree-awareness rule to candidates $\langle c_2, c_1 \rangle$ or $\langle c_2, c_3 \rangle$. In both cases, the RVAs that derive from the element table of *actor* are omitted.

2nd tree-awareness rule For the second scenario, we require the child axis of possibility nodes to not contain distributional nodes. The second scenario is illustrated in Figure 10.13c. We refer to this scenario as twig pattern p_2. We observe that p_2 is constructed as three candidates $-c$, c' and $c''-$ where c' is the parent of c and c'' is a descendant of c'. We denote an instantiation of p_2 as $\langle n, n', n'' \rangle$ such that node n instantiates candidate c, node n' instantiates candidate c' and node n'' instantiates c''. We observe that for any instantiation, n' is a common descendant of n and n''. We apply Theorem 2 in order to obtain $\Uparrow_{n'} \subseteq \Uparrow_n$ and $\Uparrow_{n'} \subseteq \Uparrow_{n''}$. Since n' is child of n, there is at most one possible choice on path $\uparrow_{n',n}$ that involves the possibility parent of n. It follows that $\#(\Uparrow_n) - \#(\Uparrow_{n'}) \leq 1$.

From this observation, we extract the following rule: let $\langle c, c', c'' \rangle$ be three candidates in a certain twig pattern p where c is a child of c' and c'' is a descendant (or child) of c'. For any instantiation of c with a node n, only its closest possible choice is added to the world set descriptor of an instantiation of p, because other possible choices of n are already added by the instantiation of c' (or c'' of the first tree-awareness rule is applied to c' and c''). One exception: if all descendants of an instantiation of c' are direct children, this rule may at most be applied to all but one children. We refer to this rule as *the second tree-awareness rule.*

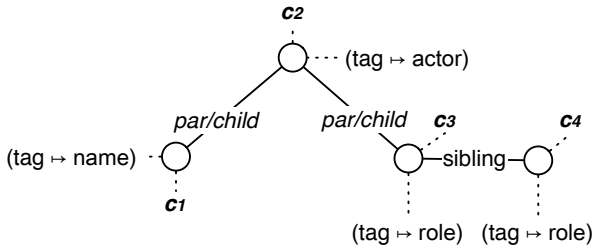
For illustration purposes, we continue with our example in Figure 10.14. We further reduce the query depth of Q_{SQL} with the second tree-awareness rule that we apply to candidates $\langle c_1, c_2, c_3 \rangle$ or $\langle c_3, c_2, c_1 \rangle$. In case of the former, we reduce the number of RVAs that are appended to the rows in the result of Q_{SQL} with 3. In case of the latter, we reduce the number of RVAs that are appended to the rows in the result of Q_{SQL} with 4. In this example, we strategically pick the reduction of 4. This gives an overall reduction of 7 RVAs on the 17 RVAs that initially needed to be appended to each row in the result of Q_{SQL}.

3rd tree-awareness rule For the third scenario, we also require the child axis of possibility nodes to not contain distributional nodes. The third scenario is illustrated in Figure 10.13d. We refer to this scenario as twig pattern p_3. We observe that p_3 is constructed as two candidates $-c$ and $c'-$ where c' is a sibling of c. From the semantics of the sibling axis, it follows that c and c' share a common parent node, also illustrated. Candidates c, c' and their common parent apply to the second tree-awareness rule.

From this observation, we extract the following rule: let c and c' be two candidates in a certain twig pattern p where c' is a sibling of c. For any instantiation of c with a node n, only its closest possible choice is added to the world set descriptor of an instantiation of p, because other possible choices of n are already added by the instantiation of c'. One exception: if all descendants of an instantiation of c' are direct children, this rule may at most be applied to all but one children. We refer to this rule as *the third tree-awareness rule.*

For illustration purposes, we continue with our example in Figure 10.14. We further reduce the query depth of Q_{SQL} with the third tree-awareness rule that we apply to candidates $\langle c_3, c_2 \rangle$. We reduce the number of RVAs that are appended to the rows in the result of Q_{SQL} with 4. This gives an overall reduction of 11 RVAs on the 17 RVAs that initially needed to be appended to each

row in the result of Q_{SQL}.



element kind	table rdepth
actor	3
name	4
role	5

Figure 10.14: Application scenario of all three tree-awareness rules

10.5 Summary

A glue process is constructed of a glue method application and a glue method. In Section 10.1, we discussed three glue method applications. Each glue method application can be combined with one of the four glue methods discussed in Section 10.2 in order to specify a glue process. We defined three categories of glue methods: flesh driven glue methods, skeleton driven glue methods and inheritance driven glue methods. Each category takes a different approach to merge the skeleton with the flesh.

The main purpose of a glue process is to associate data entries with world set descriptors (WSDs) constructed as a set of RVAs that describe the set of possible worlds of which a data entry is member. WSDs allow the query evaluation process to verify if there exists a possible world of which a query result is member. More specifically, if all data entries of which a query result derives are member of some possible world, the query result is part of that possible world as well. In order to enable the query evaluation process to use WSDs for the verification of query results, we suggest two designs:

- The first design administers a glue process to the data mapping in order to merge the flesh and the skeleton to a set of U-Relations. This design enables the evaluation of XPath queries on P-XML data as an SQL query –derived from an ordinary XPath to SQL query mapping– evaluated on a set of U-Relations.
- The second design administers a glue process to the result of a query evaluated on the flesh. This design enables the evaluation of XPath queries on P-XML data as an SQL query evaluated on flesh and skeleton. This SQL query is constructed as an ordinary XPath to SQL query mapping extended with a glue process.

Part IV

Validation

Chapter 11

Overview of MayBMS & Optimizations

11.1 Repair-key statement

MayBMS is a URDBMS that extends PostgreSQL –a traditional RDBMS– with an uncertainty management mechanism. We discussed this mechanism in detail in Chapter 4. In order to take advantage of this uncertainty management mechanism, data has to conform to the U-Relational data format. This is accomplished with the repair-key statement (f_{rk}) of MayBMS that we discuss in Sections 11.1.1 and 11.1.2. Function f_{rk} is a mandatory component of the skeleton mapping (f_{sk}). We identify a performance problem for f_{rk} in Section 11.1.3. In Section 11.1.4, we propose the multi-union approach to mitigate this problem. This generic applicable approach optimizes SQL queries that contain many union operations. We rewrite f_{rk} as an SQL query with many union operations such that it can be optimized. In Section 11.1.5, we perform a small benchmark that shows the performance benefits of using the multi-union approach to evaluate f_{rk}.

11.1.1 From inconsistency to uncertainty

The repair-key statement (f_{rk}) of MayBMS performs a repair operation on primary key violations in order to transform an inconsistent certain table into a consistent uncertain table (called a U-Relation). A primary key is a set of columns that has a unique combined value for each row such that all rows in a table can be identified with solely knowledge of the primary key columns. A primary key violation is a violation of this uniqueness constraint. Function f_{rk} repairs this kind of inconsistency as being uncertain about which of the conflicting values is the correct one. Since a table can have multiple primary key violations, we refer to each set of rows –larger than 1– with identical primary keys as a set of conflicting rows.

Primary key violations are resolved as follows. For each set of conflicting rows, each entry is associated with the same random variable identifier and a unique assignment value. It follows from the definition of RVAs that rows with the same primary key are associated with conflicting RVAs. As a consequence, there does not exists a possible world of which two rows with the same primary key are member. Additionally, non-conflicting rows are associated with different random variable identifiers such that their RVAs do not conflict. As a consequence, there does not exists a possible world that contains rows with conflicting primary keys.

For illustration purposes, we apply f_{rk} to the table 'opposites' in Figure 11.1a. This table holds a series of opposite words. Rows that represent matching opposites are given identical values in the 'id' column. We intend to use this column as primary key column. We apply f_{rk} with parameters 'id' and 'weight' to fix any primary key violation in this column. The output of f_{rk} is the original table appended with three columns. The first column holds random variable identifiers. The second column holds assignment values. Hence, the first and second column define an RVA for each row. The third column holds probabilities of RVAs. This likelihood is calculated as the evenly distributed weight over all conflicting row. We notice that conflicting RVAs are assigned to all the rows involved in a primary key conflict. Thus, the row with condition 'active' is associated with ($x \mapsto 2$) and its opposite –the row with condition 'lazy'– is associated with ($x \mapsto 3$). Since values 2 and 3 cannot be assigned to x simultaneously, there exists no possible world of which these two rows are member.

input repair-key(id,weight)

id	condition	weight	_v0	_d0	_p0
0	not conflicting	1260	a	1	1.0
1	active	504	x	2	0.4
1	lazy	756	x	3	0.6
2	above	378	y	4	0.3
2	below	882	y	5	0.7
3	before	1134	z	6	0.9
3	after	126	z	7	0.1
4	not conflicting	1	b	8	1.0

(a) Output

id	condition	weight	_W	_d0
0	not conflicting	1260	1260	1
1	active	504	504	2
1	lazy	756	756	3
2	above	378	378	4
2	below	882	882	5
3	before	1134	1134	6
3	after	126	126	7
4	not conflicting	1	1	8

(b) 'temp2'

id	_v0	weight
0	a	1260
1	x	1260
2	y	1260
3	z	1260
4	b	1260

(c) 'temp3'

Figure 11.1: f_{rk} applied to table 'opposites'

11.1.2 Implementation details

repair-key statement (f_{rk}) performed on a table 'opposite' in Figure 11.1a is internally transformed to the query in Figure 11.2. We observe two natural joins that combine the intermediate results 'temp1', 'temp2' and 'temp3'. These results are illustrated in Figures 11.1b and Figure 11.1b. Table 'temp3' holds random variable identifiers in column '_v0', one for each unique primary key value. Table 'temp3' is obtained as a 'GROUP BY'-expression on 'id'. The total weight of each set of conflicting rows is calculated and stored in column '_S'. Intermediate result 'temp2' holds unique values for each row in column '_d0'. Column '_W' is a copy of the defined weight column.

The final result of f_{rk} enriches the original input –stored in intermediate result 'temp1'– with the use of two natural joins[1] in order to (1) associate rows that have identical primary key values with the same random variable identifier stored in column '_v0', (2) associate each input row with a unique value in column '_d0', and (3) calculate the likelihood of each row as its weight divided by the total weight of the set of conflicting rows it is member: '_W' / '_S'. In case a row does not conflict, it has a likelihood of 1.

11.1.3 Small benchmark on repair-key statement

The query execution plan (not shown in this thesis) of Figure 11.2 shows four sort operations performed on the 'id' column of table 'opposite'. Sort operations tend to be expensive operations in terms of execution time. We suspect the performance of f_{rk} to be largely dependent on the performance of these sort-operations.

We assume sort operations to have the best performance on data sets for which all entries have the same value. We define our first test set *sameKey* as (1) a set of data sets that increase in data set size and (2) for which all entries in one data set are identical. We assume sort operations to perform poorly on data sets for which all entries have different sort values (the value on which the sort is performed). We define our second test set *differentKey* as a (1) set of data sets that increase in data set size and (2) for which all entries in one data set have different sort values. Based on previous assumptions, we identify *sameKey* to represent the best case scenario for f_{rk} and *differentKey* to represent the worst case scenario. We assume the execution time of any other data set to be bound by the execution times of f_{rk} on *differentKey* and *sameKey*.

Figure 11.3 shows the benchmark results for increasing data set size for test sets *differentKey* and *sameKey*. The X-axis shows the average execution time per table entry. Each result is obtained with 10 executions on a hot database. We observe that the execution time to process an entry in *differentKey* deteriorates while the execution time to process an entry in *sameKey* remains constant. We observe that the results on *differentKey* show an optimal execution time per entry for data sets between the 400 and 700 entries while the results on *sameKey* appear to be optimal for data sets with more than 900 entries.

[1] A natural join performs an equality check on the columns that have identical column-names. The resulting joined table contains one column for each pair of equally named columns.

```
SELECT
  temp1.id,                          -- original input column
  temp1.condition,                   -- original input column
  temp1.weight,                      -- original input column
  (temp3._v0)::integer AS _v0,       -- column with random variable names
  (temp2._d0)::integer AS _d0,       -- column with random variable assignment values
  (temp2."_W" / temp3."_S") AS _p0   -- likelyhood of the random variable assignment being correct
FROM
(
  (
    -- Original table
    (SELECT t.id, t.condition, t.weight FROM ONLY opposites t) temp1
    NATURAL JOIN
    (
      SELECT DISTINCT
        temp.id, temp.condition, temp.weight,
        test_negative(test_negative((temp.weight)::real)) AS "_W", -- copy of the weight column with a non_negative weight
        nextval('domid'::regclass) AS _d0  -- generates id value with a sequence generator of PostgreSQL
      FROM (SELECT t.id, t.condition, t.weight FROM ONLY opposites t) temp
      WHERE (test_negative((temp.weight)::real) <> (0)::double precision)
      GROUP BY temp.id, temp.condition, temp.weight
      ORDER BY temp.id, temp.condition, temp.weight, test_negative(test_negative((temp.weight)::real)),
nextval('domid'::regclass)
    ) temp2
  ) NATURAL JOIN (
    SELECT DISTINCT
      temp.id, -- primary key value
      nextval('varid'::regclass) AS _v0, -- generates identifiers for random variables
      sum(temp."_W") AS "_S"  -- total weight of all rows with the same primary key value
    FROM (
      SELECT DISTINCT temp.id, temp.condition, temp.weight, test_negative(test_negative((temp.weight)::real)) AS "_W"
      FROM (SELECT t.id, t.condition, t.weight FROM ONLY opposites t) temp
      WHERE (test_negative(test_negative((temp.weight)::real)) <> (0)::double precision)
      GROUP BY temp.id, temp.condition, temp.weight
      ORDER BY temp.id, temp.condition, temp.weight, test_negative(test_negative((temp.weight)::real))
    ) temp
    GROUP BY temp.id
    ORDER BY temp.id, nextval('varid'::regclass), sum(temp."_W")
  ) temp3
);
```

Figure 11.2: f_{rk} – under the hood

11.1.4 Multi-union approach

The performance study in the previous section gave rise to a new approach to evaluate f_{rk}. We refer to this approach as the *multi-union* approach. The multi-union approach pushes f_{rk} through the union operator such that the result of f_{rk} remains the same. In more formal terms, the following applies:

$$f_{rk}(ds) = f_{rk}(\bigcup\{d_1, \ldots, d_n\}) = \bigcup\{f_{rk}(d_1), \ldots, f_{rk}(d_n)\} \text{ for } \langle d_1, \ldots, d_n\rangle \text{ partition } ds$$

Effects of f_{rk} We have the obligation to show that f_{rk} pushed through the union operator does not affect the result. Therefore, we first summarize the effects of f_{rk}. Effects of f_{rk} on a high level: (1) associate conflicting rows with conflicting RVAs and (2) associated non-conflicting rows with non-conflicting RVAs. Figure 11.2 shows the implementation of f_{rk}. Effects of f_{rk} on a low level:

1. Rows with the same primary key are associated with the same random variable.
2. Rows with the same primary key are associated with a different assignment value.
3. The probability of a row to be in a possible world is calculated as its own weight divided by the total weight of the set of conflicting rows it is member of.
4. Rows with a different primary key are associated with a different random variable.

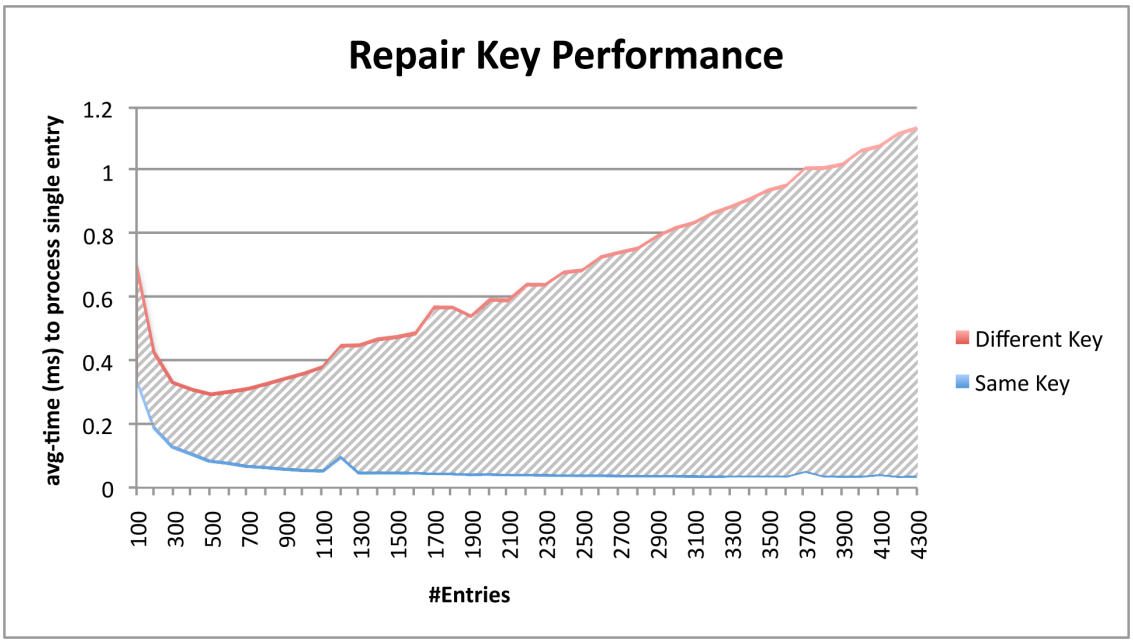


Figure 11.3: Performance of repair-key statement for increasing data set size

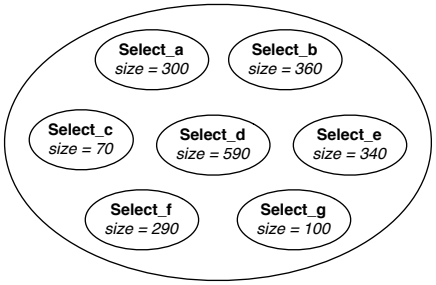
Partitioning requirements We specify a set of partitioning rules such that the effects of f_{rk} pushed through the union operator is unaffected. We have to show that:

1. Each two conflicting rows are associated with the same random variable.
2. Each two conflicting rows are associated with a different assignment value.
3. The same probabilities are assigned to rows.
4. No non-conflicting rows are associated with the same random variable.

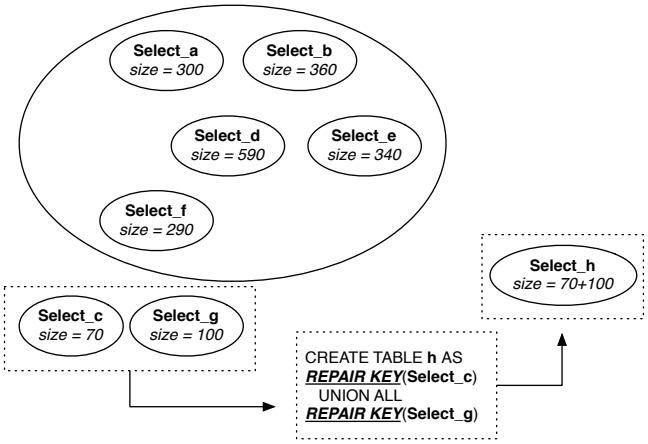
It follows from the implementation of f_{rk} that the first three requirements are met if all sets of conflicting rows are subsets of partitions. The fourth requirement is met if different random variables are created for multiple evaluations of f_{rk}. Fortunately, the implementation of f_{rk} uses a one sequence generator to generate random variable identifiers and one sequence generator to generate assignment values. According to the PostgreSQL manual [1], sequence generators are globally known to the database and each generator generates $2^{63} - 1$ unique values by default. Therefore, the first $2^{63} - 1$ created random variables are unique. We conclude that the fourth requirement is satisfied.

Optimal partitioning size Our second goal is to identify how large multi-union partitions should be such that we obtain optimal performance for f_{rk}. We have to keep in mind that the smaller the partitions, the more merge-operations have to be executed. Benchmark results in Figure 11.3 show performance of f_{rk}. We assumed the red line to define an upper bound and the blue line to denote a lower bound for the execution times of f_{rk}. We observe the upper bound to be minimal for data set sizes between the 500 and 700 entries. We select a partition size of 600 entries. Slight deviations are allowed to ensure that all sets of conflicting rows are subsets of partitions.

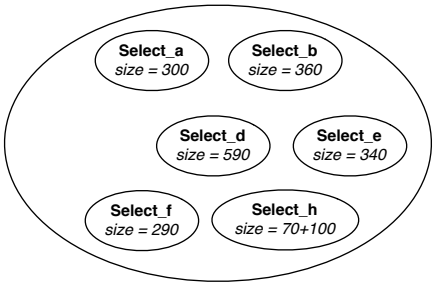
Most efficient merge ordering If we apply f_{rk} to various partitions, we obtain partial results that have to be merged into one final result. We define a merge ordering such that the merge process is accomplished as efficiently as possible. We note that MayBMS does not support SQL-statements with more than one UNION-operation. Hence, we are obliged to store intermediate merge results. These results are used as input for subsequent UNION-operations until all intermediate results are



(a) Input: a set of select statements with an expected size.



(b) Union the two selects with the lowest expected size.



(c) Put a select of the union result back in the pool.

Figure 11.4: Multi-union approach example.

processed into one fully-merged result. We are free to choose in what order we merge intermediate results. The UNION-operator performs best on small data sets. Therefore, we define a merge ordering as a series of steps that performs UNION-operations on the smallest operands. These steps are also illustrated in Figures 11.4a, 11.4b, 11.4c.

1. Define a pool initially filled with unprocessed partitions of which the size of each partition is known;
2. Retrieve the two smallest pool entries. Either an entry is an unprocessed partition or an intermediate result. In case of the former, apply f_{rk}. Perform a merge-operation with the selected pool entries.
3. Put the result of the merge-operation back in the pool as an intermediate result such that it can be used as input for a following merge-operation;
4. If the pool only contains one entry, that entry is the result of the multi-union approach; else, return to step 2.

Figure 11.5 presents a binary processing tree that illustrates the processing of Figure 11.4a. Partitions are the leaves in this binary tree; arrows denote operands for a merge-operation; the result is the root of the binary tree. Note that each partition is processed by f_{rk} before it is used as an argument of a merge-operation.

Optimize PostgreSQL with merge ordering The manual of PostgreSQL [1] states that a statement with multiple UNION-operations are evaluated in sequential order. In our opinion, PostgreSQL can optimize SQL-statements that use multiple UNION-operations with our merge ordering.

11.1.5 Benchmark on multi-union approach applied to repair-key

In Chapter 12, we perform several experiments with various P-XML into URDBMS mappings that all make use of multi-union approach to evaluate f_{rk} as part of their skeleton mapping. In this section, we present a small benchmark that shows the performance improvements as a result of the multi-union approach. We performed this benchmark on our test set of increasing data set size. This test set is in more detail described in Section 12.2.1. We obtained the benchmark results in a similar fashion as described in Section 12.2.5.

Figure 11.6 shows the results of our experiments for increasing amounts of size. We observe a linear increase of speedup. This indicates a linear reduction in execution time. Note that the y-axis shows the actual speedup in terms of how my times one algorithm is faster than another. The performance benefits of the multi-union approach are quite large.

We conclude that the evaluation of f_{rk} with the multi-union approach is desirable. Experimental results in Section 12.3.1.2 indicate a linear correlation between number of distributional nodes and execution costs of the repair-key statement performance with the multi-union approach.

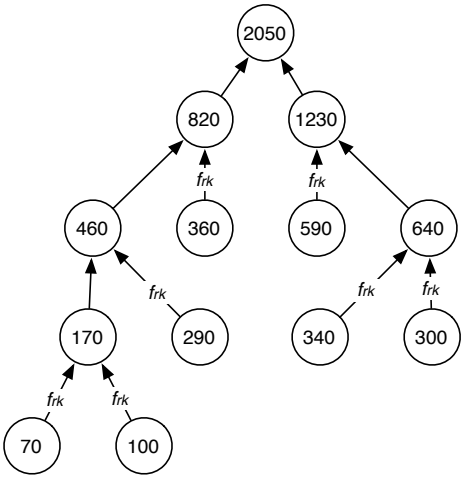


Figure 11.5: Binary tree that visualized the merge ordering

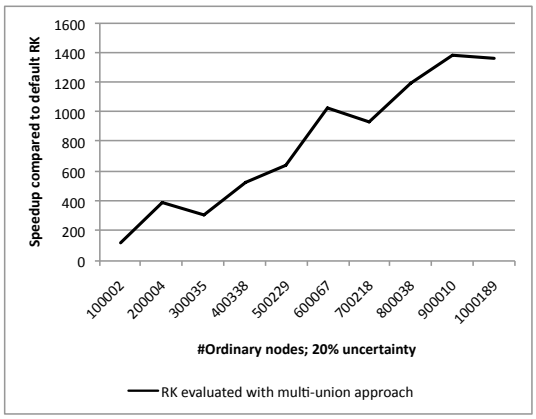


Figure 11.6: Test set for increasing data set size

11.2 XPath Accelerator vs. Abstract Shared Inlining

In Section 10.3.1, we introduced the concept of document oriented gluing (DO). A DO glue process is part of the data mapping and combines the result of flesh mapping (f_{fl}) and skeleton mapping (f_{sk}) such that the result is a set of rows appended with RVAs. We consider the result of f_{fl} to be a set of relational tables ts_{fl} that originate from a p-document pd. A DO glue process is applied to each of the tables in ts_{fl} such that all rows in $t \in ts_{fl}$ are appended with rdepth(t) RVAs.

In this section, we show that for any DO glue process, the overall number of RVAs appended to tables in ts_{fl} is reduced if ASI[XA] serves as f_{fl} instead of XA to a p-document pd. We assume pd to have (1) a certain depth depth(pd), (2) a total of n_{text} text nodes, and (3) a total of n_{xml} XML nodes.

Storage costs of DO with XA XA maps an XML-document to the tables *accel* and *contents* as described in Section 2.3.2. If we use XA as f_{fl}, we write $ts_{fl} = \{accel, contents\}$. Table *accel* represents all ordinary nodes in pd represented as 5-tuples. A DO glue process applied to *accel* associates depth(pd) RVAs to all rows in *accel* since rdepth($accel$) = depth(pd). As a consequence, the WSD of each row is represented as a $3 \times$ depth(pd)-tuple. Based on the previous, we calculate the storage costs of *accel* as its number of row-attributes:

$$C_{accel} = (n_{xml} + n_{text}) \times (5 + 3 \times \text{depth}(pd))$$

Table *contents* represents the PCData of text nodes as 2-tuples. It is likely that rdepth(*contents*) is close to depth(pd). This follows from the fact that the *contents* table represents all text nodes in pd which reside in the lower levels of the document. For these lower levels holds that depth is likely to be close to the maximum. We assume rdepth(*contents*) = depth(pd). Based on the previous, we calculate the storage costs of *contents* as its number of row-attributes:

$$C_{contents} = n_{text} \times (2 + 3 \times \text{depth}(pd))$$

We define the total storage costs of DO with XA as $C_{XA} = C_{accel} + C_{contents}$.

Storage costs of DO gluing with ASI[XA] ASI[XA] maps an XML-document to a set of element tables. If we use ASI[XA] as f_{fl}, we write ts_{fl} to refer to this set of element tables. In order to calculate the storage costs, we assume that text-nodes are the only nodes that are inlined. Furthermore, we consider the storage costs of a data dictionary to be negligible. In Section 8.4.2, we define ASI[XA] to represent non-inlined XML nodes as 3-tuples and inlined text nodes as 1-tuples. A DO glue process applied to table $t \in ts$ associates all rows in t with m RVAs. The amount of m is determined by the row in t that represents the node with the largest depth. Hence, m is at most depth(pd). In this case, there exists a row in t that represents a node at the deepest depth of pd. In worst case scenario, all tables in ts have a row that represents a node at the deepest depth. However, this is highly unlikely in practice. Our real world uncertain test set shows that the average rdepth of tables in t is 0.513 $\times$depth(pd) We assume this figure to be representative. We calculate number of row-attributes of a DO glue process applied to the element tables of ASI[XA] as:

$$C_{ASI[XA]} = (n_{xml} \times (3 + 3 \times 0.513\,\text{depth}(pd)) + (n_{text} \times 1)$$

Since we assumed text nodes to be inlined, they share the WSD with the XML node they are inlined with.

Storage costs of XA compared to ASI[XA] We rewrite C_{XA} and $C_{ASI[XA]}$ as a formula of n_{xml} and n_{text} in order to compare storage costs to represent XML nodes and storage costs to represent text nodes. We note that some assumptions we made in order to define $C_{ASI[XA]}$ may not hold in practice. However, the definitions $C_{ASI[XA]}$ and C_{XA} estimate the number of

row-attributes pretty well for the test sets we describe in our experimental section.
Storage costs of a DO with XA

$$\begin{aligned} & C_{XA} \\ &= (n_{xml} + n_{text}) \times (5 + 3 \times \text{depth}(pd)) + n_{text} \times (2 + 3 \times \text{depth}(pd)) \\ &= n_{xml}(5 + 3 \times \text{depth}(pd)) + n_{text}(5 + 3 \times \text{depth}(pd) + 2 + 3 \times \text{depth}(pd)) \\ &= n_{xml}(5 + 3 \times \text{depth}(pd)) + n_{text}(7 + 6 \times \text{depth}(pd)) \end{aligned}$$

Storage costs of a DO glue process applied to ASI[XA]

$$\begin{aligned} & C_{ASI[XA]} \\ &= n_{xml}(3 + 3 \times 0.513\,\text{depth}(pd)) + n_{text} \end{aligned}$$

We conclude:

- The costs to represent an XML node with world set descriptor is $(5 + 3 \times \text{depth}(pd))$ row attributes when the XA approach is used compared to $(3+3\times0.513\,\text{depth}(pd))$ row attributes when the ASI[XA] is used.

- The costs to represent a text node with world set descriptor is $(7 + 6 \times \text{depth}(pd))$ when the XA approach is used compared to 1 row attribute when the ASI[XA] is used.

We conclude that ASI[XA] represents P-XML data more efficiently. Furthermore, it stands to reason that evaluation costs of DO glue processes are also reduced since the don't care RVA is assigned lesser times.

For illustration purposes, we make the same storage costs comparison based on our running example in Figure 2.8. First, we calculate the storage costs of XA. We count 14 XML nodes, 3 text nodes and observe a document depth of 3. If we use these numbers as input for C_{XA}, we obtain 271 row attributes as storage costs for XA Second, we determine the storage costs of ASI[XA]. The flesh result of ASI[XA] is illustrated in Figure 10.11. This figure shows 9 element tables constructed of 36 row attributes. Therdepth value of each element table is denoted at the bottom of each element table and is consistent with Figure 2.8. We observe that 1 row requires no RVAs, 4 rows require 1 RVA, 2 rows require 2 RVAs and 4 rows require 3 RVAs. In total, 20 RVAs are appended. We obtain 56 row attributes as storage costs for ASI[XA].

Depth of a query result The storage costs comparison between XA and $ASI[XA]$ shows that the number of don't care RVAs is reduced when ASI[XA] is used. It follows from the definition of the depth of an SQL query that the query results are reduced in size as well.

As an additional advantage, the time it takes to verify all world set descriptors of rows in a query result is likely to decrease if less don't care RVAs have to be checked.

11.3 Glue methods

We defined four different glue methods in Chapter 10. We implemented each of these glue methods in SQL. The resulting SQL code is found in Appendix B.

Glue by Possibility Parent Reference Glue method PPR is designed as a pointer from a phase table to the skeleton. Appendix B.1 shows SQL query q_{ppr} that is used to evaluate on BB.SW phase. BB.SW is a glue process constructed of glue method PPR and glue method application BB.

Query q_{ppr} takes phase table ph and skeleton sk in order to generate the phase table that follows ph. Column $ph.posspre$ stores pointers that match entries in $sk.pre$.

Glue by Sandwich Glue method SW is designed as a complex expression that returns the closest ancestor choice. Appendix B.2 shows this expression as SQL query q_{sw} that is used to evaluate on BB.SW phase. BB.SW is a glue process constructed of glue method SW and glue method application BB.

Query q_{ppr} takes phase table ph and skeleton sk in order to generate the phase table that follows ph. The first two rows of the WHERE-clause select for each entry e in ph a subset $as \subseteq sk$ that are ancestors of e. The next part of the query selects the entry in as that is closest to e. We refer to this one entry as a. For a holds that there exists not another $a' \in sk$ that is ancestor of e and descendant of a.

We construct the nested query of q_{ppr} as the COALESCE-function. This function returns the first of its arguments that is not null. Null is returned only if all arguments are null. We construct the second argument of the COALESCE-function as a reference to the don't care RVA. Thus, if an entry in ph does not have ancestors in sk, the COALESCE-function associates the don't care RVA with that entry.

Glue by Closest Descendant Glue method CD is designed as an aggregate that returns the closest ancestor choice. Appendix B.3 shows this expression as SQL query q_{cd} that is used to evaluate on BB.CD phase. BB.CD is a glue process constructed of glue method CD and glue method application BB.

Query q_{cd} takes phase table ph and skeleton sk in order to generate the phase table that follows ph. The first two rows of the WHERE-clause select for each entry e in ph a subset $as \subseteq sk$ that are ancestors of e. The next part of the query selects the entry in as that is closest to e. We refer to this one entry as a. For a holds that its pre-order is highest of all entries in as.

Analogously to SW, we construct the nested query of q_{ppr} as the COALESCE-function in order to match entries in ph with the don't care RVA in case no other entry in ph matches.

Glue by Depth Glue method D is designed as an expression that finds all descendants to a set of skeleton entries. Appendix B.4 shows this expression as SQL query q_d that is used to evaluate on BB.D phase. BB.D is a glue process constructed of glue method D and glue method application BB.

Query q_d is constructed as a UNION-operation. Its first operand matches entries in sk with a depth of x to their descendants. The second operand assigns the don't care RVA to all entries in ph that are not descendants of entries in sk with a depth of x.

Chapter 12

Experiments

12.1 Goal

Our experimental goal is twofold. We are interested in (1) the scalability and performance of the P-XML into URDBMS data mappings discussed in this thesis, and (2) the efficiency of XPath evaluation with a URDBMS.

Our designs of a P-XML into URDBMS data mapping are use the following parts: (1) a mapping of the flesh, (2) a mapping of the skeleton, (3) the construction of skeleton path tables, and (4) a DO glue process. We conduct a performance study on all four parts in order to derive the scalability and performance of P-XML into URDBMS data mappings.

Our second experimental goal focuses on the efficiency of XPath evaluation, in particular, (1) the performance of t-queries compared to (2) the performance of tg-queries. Both query kinds are discussed in Section 10.3. A performance study has to point out how well these two areas of interest scale for increasing amounts of uncertainty and increasing data set size.

A glue process is administered to the data mapping or to the query mapping. In either case, one of the experimental goals is affected. In Chapter 10, we define a glue process as an application of a glue method. We are interested in the behaviour of both the selected type of application and the selected type of glue method regardless of the administering in order to identify the effects of a glue process on either the performance of a data mapping or the efficiency of XPath evaluation by a URDBMS.

QRO glue processes that use glue method CD or SW allow for a loosening of the inline rules such that the the more scenarios profit from inlining. We are interested to what extent query performance profits from a loosening of the inline rules.

The kind of axes used in an XPath expression and the complexity of XPath predicates is likely to influence query execution time. Therefore, we categorize our query workload in order to investigate the effects of the kind of XPath expressions evaluated with our approach.

12.2 Experimental setup

The proposed designs for P-XML into URDBMS database mappings raises scalability and feasibility questions. We developed a prototype using Java JDK 1.6 and MayBMS version 2.1-beta on top of PostgreSQL version 8.3.3 in order to answer these questions. This prototype maps p-documents using the different glue processes discussed in Chapter 10.

12.2.1 Test sets

We consider a *test set* to be a set of data sets for which one property is varied. A data set is a set of data on which a query workload is directly evaluated. In our case, single p-documents. All test sets are derived from the original IMDb data set; a real world XML data set in a proprietary format from the Internet Movie Database (`http://www.imdb.com`). Note that this data set does not contain any uncertainty. The IMDb data set is a balanced 5-level-deep tree. We derive two synthetic test sets from the IMDb data set: (1) a test set for increasing amounts of uncertainty, and (2) a test set for increasing data set size. Additionally, we use a real world uncertain test set (also derived from the same IMDb data set).

Generate uncertainty by transforming XML-documents into p-documents We transform XML-documents into p-documents with a top-down approach we refer to as δ. Process δ requires a fixed probability parameter p_δ. For each node n processed by δ, children of n are

randomly selected based on p_δ. We refer to the set of selected children of n as ns. Next, a new probability node v is assigned to n as a child and each edge between n and $n' \in ns$ is replaced with a possible choice $(v \mapsto d)$ such that n' is put in the child axis of d and n is put in the parent-axis of v. An illustration of this approach is found in Figure 12.1: a node n_a is processed by which all children of n_a are examined. The children n_{a2} and n_{a3} are selected. The edge between these two nodes and their parent n_a is replaced with a possible choice.

We do not process all nodes in a p-document. We start with the root node and only process selected children. This is illustrated in the right most picture of Figure 12.1. Process δ start with processing the root node by which two children are selected that reside at the first level. In following levels, selected nodes are in all cases children of other selected nodes that reside at a higher level. Nodes at the first level of an XML-document have a likelihood of p_δ to be selected; nodes at the second level have a likelihood of p_δ^2 to be selected; etc. As a consequence, we create hotspots of uncertainty in our resulting document and large parts without uncertainty. This is in line with the uncertainty distributions in real world documents.

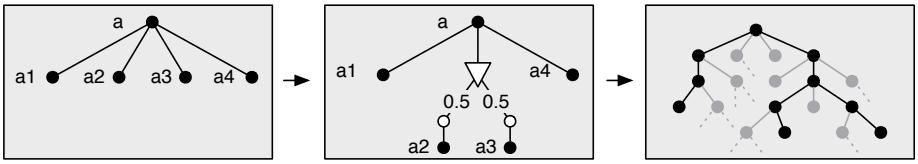


Figure 12.1: Generate uncertainty

In order to compare the amount of uncertainty in different p-documents, we define the *uncertainty ratio* of a p-document as the number of ordinary nodes divided by the number of possibility nodes. The higher the ratio, the less uncertainty in a p-document The number of possibility nodes is representative for the number of choices in a p-document. Hence we express the uncertainty ratio as the ratio between ordinary nodes and possibility nodes (instead of ordinary nodes and distributional nodes).

If we apply δ to the first 10^5 nodes of the IMDb data set and we set off p_δ against the number of created possibility nodes, we obtain the graph in Figure 12.2a. This graph shows perfect quadratic increasing behaviour. Additionally, we add Figure 12.2b to show the effects of p_δ on the uncertainty ratio.

Test set of increasing amounts of uncertainty We take a subset of the first 10^5 nodes from our previously mentioned IMDb data set. We apply δ with parameter p_δ varied from 0.1 to 1.0 in nine steps. We consider these 10 uncertain data sets to be the test set of increasing amounts of uncertainty. The amount of ordinary nodes is the same in all data sets. The amount of possibility nodes increases quadratically as shown in Figure 12.2a

Test set of increasing data set size We take 10 subsets from our previously mentioned IMDb data set with sizes ranging from 10^5 nodes to 10^6 nodes linearly increased in 9 steps. We apply δ with parameter $p_\delta = 0.2$. We consider the resulting 10 data sets to define the test set of increasing data set size. The uncertainty ratio of these 10 data sets is 61 ordinary nodes per possibility node (maximum deviation is 3). It follows from the definition of uncertainty ratio that the number of possibility nodes increases linearly.

A real world uncertain test set Van Keulen et al. [50] experimented with a probabilistic approach to enrich a top-100 of the TV-Guide with information from an IMDb data set; the same data set as we described earlier. They use a "movie title threshold" and a margin to match information from both sources. The result of their experiments are p-documents with varying amounts of uncertainty. More specifically, each combination of movie title threshold and margin results in one p-document. We use this collection of p-documents as third real world test set.

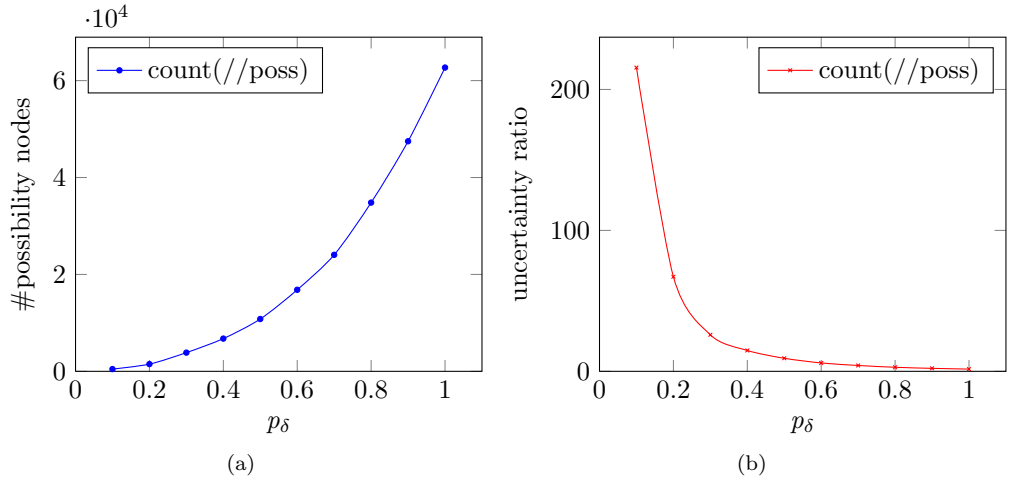


Figure 12.2: Behaviour of generating uncertainty

Experimental results in Figure 17a of [50] show that a lower movie title threshold results in an increase of nodes in the resulting p-document. The margin does not seem to influence the number of nodes in Figure 17a. Therefore, we use the movie title threshold as the property we vary in this test set. In order to ignore margin, we take the average execution time of queries executed on data sets with identical movie title threshold in future benchmarks.

The uncertainty ratio of all data sets in this real world uncertain test set is 7.1 ordinary nodes per possibility node (maximum deviation is 0.3).

12.2.2 Query workload

Test queries need to be carefully selected for a performance study. We select 23 XPath queries as workload to cover most frequently used queries and most aspects of XPath. All of these queries are found in Appendix C.1. We categorize queries in 5 categories:

- *Complex Predicates*: we select 6 XPath expressions that request movies for which a certain predicate has to hold. This predicate requires multiple descendants of that movie to satisfy a certain condition. In general, this kind of predicates are efficiently evaluated with nested loops. We characterize these predicates as complex. *Example:*
  `//movie[(.//actor/name/string() = //director/string())]/title`

- *Preceding-sibling & following-sibling*: we select 4 XPath expressions to cover the preceding-sibling & following sibling axis. Each following query adds another location step with sibling axis to its preceding query. Note that the latter three expressions request the same result. *Example:*
  `//movie/descendant::genre[string() = $genre]/following-sibling::genre`

- *Single type ancestor & descendant*: we select 4 XPath expressions to cover the ancestor & descendant axis. The first and third expression request the same result. Analogously the second and fourth expression. *Example:*
  `//genre[./string() = $genre]/ancestor::movie/descendant::title`

- *Parent & child*: we select 4 XPath expressions to cover the child & parent axis. Like in the previous category, the first and third expression and the second and fourth expression request the same result. *Example:*
  `//actor[name/string()=$actor_name]/child::role/parent::*`

- *Simple predicates & varying axes*: we select 5 XPath expressions to cover tree-traversals with varying XPath axes and simple predicates. *Example:*
  `//actor[name/string()='Mehaffey, Blanche']/ancestor::*`

12.2.3 Database optimizations

Queries derived from the query workload –discussed in the previous section– can become very complex. In order to guide PostgreSQL to pick the most efficient query execution plan, we had two tools at our disposal: (1) a CREATE INDEX statement to create B-tree indices, either clustered or unclustered, and (2) SET operators [1] to enable or disable (2.1) sequence scans, (2.2) nested loops, (2.3) merge joins, and (2.4) hash joins.

In order to find the optimal set of indices and set operators for each test query, we made an attempt to evaluate each possible combination. We accomplished this for indices as follows. First, we extracted all columns from the SQL variant of a test query and grouped these columns by associated table. Second, we take the permutation of each group of columns. Each permutation holds all orders for used columns per table. Third, we pick one column order from each permutation to build an index. The set of indices built with all possible picks composes the set of all possible indices.

We managed to find the optimal set of indices and set operators for queries 1.1-1.3, 1.5, 2.*,3.*,4.* and 5.*. The search space for queries 1.4 and 1.6 was to large to process completely. In this search process, we did not take clustered B-tree indices into account. The result of the search process is presented in Section C.2.

12.2.4 Test platform

All testing is performed on a MacBook Pro with 4 GB 1067 Mhz DDR3, 3.06 GHz Intel Core 2 Duo processor, 256GB SSD hard drive (APPLE SSD TS256A) and Mac OS X version 10.6.8 as operating system. There were no other concurrent processes running during the experiments besides a small number of sleeping system daemons.

12.2.5 Obtaining benchmark results

Determine execution time We used the default PostgreSQL timing mechanism to retrieve query execution times. This mechanism is enabled with the command '\timing'. We measured the time it takes to execute the result of an SQL query mapped from an XPath query. The time it takes to map an XPath query into an SQL query is not taken into account, because we are only interested in the performance of the URDBMS. For query execution benchmarking, we use the average execution time over the last 100 runs. We executed 101 runs to ensure the database was hot.

The time to create indices is included in the execution time for the data mapping. Indices created for element tables are included in the execution time to map the flesh; indices created for phase table are included in the execution time of a DO glue process. In particular, BB glue processes create intermediate indices.

Normalized execution times In order to give a fair comparison between the performance of different approaches that evaluate a set of varying expressions, we normalize execution results. We explain normalization by example. Let a_1, a_2, a_3 be three algorithms to evaluate a certain query q and let $t_1(q), t_2(q), t_3(q)$ be the query execution times of each of the three methods. We normalize these query execution times with a function n as follows:

$$n(t_1(q)) = t_1(q)/\sum\{t_1(q), t_2(q), t_3(q)\}$$
$$n(t_2(q)) = t_2(q)/\sum\{t_1(q), t_2(q), t_3(q)\}$$
$$n(t_3(q)) = t_3(q)/\sum\{t_1(q), t_2(q), t_3(q)\}$$

Queries	T(a_1)	T(a_2)	T(a_3)	NT(a_1)	NT(a_3)	NT(a_3)
q_1	12	12	3	0.44	0.44	0.11
q_2	6	12	6	0.25	0.50	0.25
q_3	31	6	65	0.31	0.06	0.65
sum	***49***	***30***	***74***	***1***	***1***	***1***

Default speedup of a_1 compared to a_2: ***0.61***
Default speedup of a_1 compared to a_3: ***1.51***

Figure 12.3: Normalized speedup compared to default speedup

In more general, for query execution times $a, \ldots, z$ derived from the same query, we obtain their normalized exection times as follows:

$$n(a, b, \ldots, z) = (a/s, b/s, \ldots, z/s) \text{ where } s = a + b + \cdots + z$$

An advantage of normalizing execution times is that the overall performance of an algorithm is not ruined by one outlier. For example, if two algorithms a_1 and a_2 have the same performance for three queries and a third algorithm a_3 evaluates the first query in twice the execution time as a_1 and a_2 but evaluates the other two queries four times as fast, all three algorithms would have a total normalized execution time of 1 regardless of the actual execution time.

In general, algorithms that manage to execute a certain query far more efficient than the average execution time of that query are rewarded. The algorithm with the lowest total normalized execution time has the best performance for queries that are considered to be hard to evaluate for the majority of the algorithms with which the winning algorithm is compared.

Speedup The concept *speedup* refers to how much a certain algorithm a is faster than a corresponding equivalent algorithm a'. Speedup is defined as the execution time of the former algorithm $T(a)$ divided by the execution time of the latter algorithm $T(a')$.

Normalized speedup We define the concept *normalized speedup* as the normalized execution time of an algorithm a divided by the normalized execution time of another algorithm a'. Like the default definition of speedup, normalized speedup is a measure of how much an algorithm a is faster than a corresponding algorithm a'. Normalized speedup and default speedup give the same result for execution times of individual queries, because normalization divides the execution times by a common denominator. For a set of query execution results, default speedup disfavors algorithms that have outliers. We illustrate this with Figure 12.3. Observe that the normalized speedup of all combinations of algorithms a_1, a_2 and a_3 is 1. This indicates that a random picked query in $\{q_1, q_2, q_3\}$ is evaluated by all three algorithms even efficient. However, the absolute difference in execution time for query 3 result in a favoring of algorithm a_2 while this algorithm is the slowest for queries 1 and 2.

Speedup should be used if a query workload is representative for default user behaviour with the restriction that each query should be assigned with a certain weight that captures how often that query is used compared to others. In contrast, normalized speedup should be used if a query workload contains queries that are likely to be evaluated by the system, however, it is unknown if the query workload gives a proper reflection of user behaviour.

Materialized vs. unmaterialized Materialized views and unmaterialized views are discussed in Section 2.6. In our experiments, we make a distinction between queries for which we evaluate the flesh result with a materialized view –denoted with postfix (mv)– and queries for which we evaluate the flesh result with an unmaterialized view –denoted with postfix (v).

For queries with the postfix (mv), we force the PostgreSQL query planner to calculate the flesh result prior to the QRO glue process. This restriction does not hold for queries with the postfix (v). As a consequence, a wider set of query plans apply to unmaterialized views, including the

execution plan to evaluate the flesh result prior to the query result glue process. Therefore, we expect queries with the postfix (v) to have a better performance than queries with the postfix (mv). In our opinion, each experimental result that shows a better performance for materialized views than unmaterialized views is an indication that an inefficient query execution plan is selected.

12.3 Experimental results

As described in Section 12.1, experiments focus on two areas of interest: (1) experiments that give insight in the scalability of P-XML into URDBMS data mappings, and (2) experiments that give insight in the performance of XPath evaluation by a URDBMS.

We abbreviate glue processes as an application followed by a glue method: $X.Y$ denotes the glue process that applies a glue method X with application flavour Y. Furthermore, if we write "a X glue process", we refer to a glue process that uses X as glue method application. Analogously, if we write "a Y glue process", we refer to a glue process that uses Y as glue method. We make a distinction between CD and CD_*TI* that both denote the glue method CD: the former uses the extended set of inlining rules while the latter uses the traditional set of inlining rules. A performance difference between the two indicates a benefit or drawback as a result of inlining.

12.3.1 Experimental results for P-XML into URDBMS data mappings

The experimental results for P-XML into URDBMS data mappings consists of two experiments. The first experiment aims to find the best application flavour for a particular glue method. The second experiment investigates which P-XML into URDBMS data mapping is the most efficient.

12.3.1.1 Performance study on application flavours

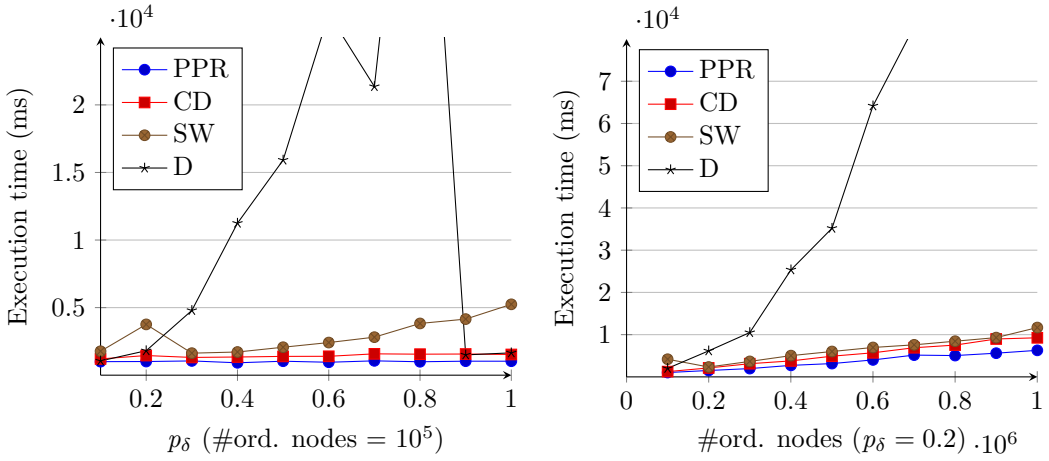
In Section 10.1, we discuss three glue method application flavours to apply a glue method: precomputed chaining (PC), batch based application (BB) and partition based application (PB). We tested these three applications for the four glue methods discussed in Section 10.2: glue by possibility parent reference (PPR), glue by closest dependency (CD), glue by sandwich (SW) and glue by depth (D). We present the experimental results for each combination of a glue method with an application flavour.

Experimental results for precomputed chaining The behaviour of the PC application is shown in Figure 12.4a for increasing amounts of uncertainty and in Figure 12.4b for increasing data set size. We make the following observations:

- For the glue methods PPR and CD, we observe constant execution time for increasing amounts of uncertainty and a slight linear increase in execution time for increasing data set size.
- For glue method SW, we observe a slight linear increase in execution time for increasing data set size and increasing amounts of uncertainty.
- For glue method D, we observe exponential behaviour for increasing data set size. We observe the same behaviour for increasing amounts of uncertainty until 60%. However, we observe a drop in execution time for 90% and 100% of uncertainty.

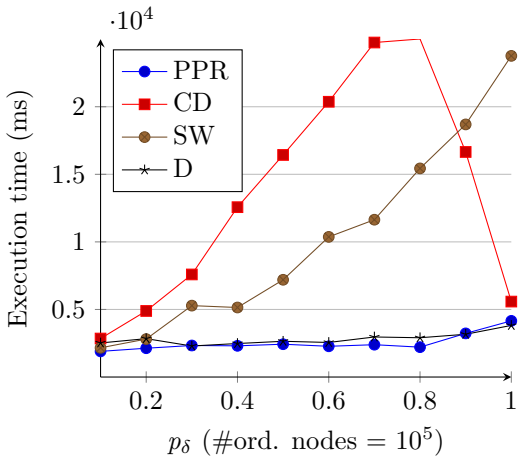
The behaviour of glue method D is likely to be caused by the distribution of the depth property of rows in the flesh. We explain this conjecture with the following. The U-Relations derived from our data sets show that for relative small amounts of uncertainty and for relative large amounts of uncertainty, the majority of rows have the same depth. The query plan of glue process PC.D selects a sort operation on the depth property of element tables in order to perform a glue process. This sort process is evaluated more efficiently for a small range of depth property values.

PC performs best in combination with the PPR glue method. The performance of PC in combination with CD is equivalent with the performance of PPR.

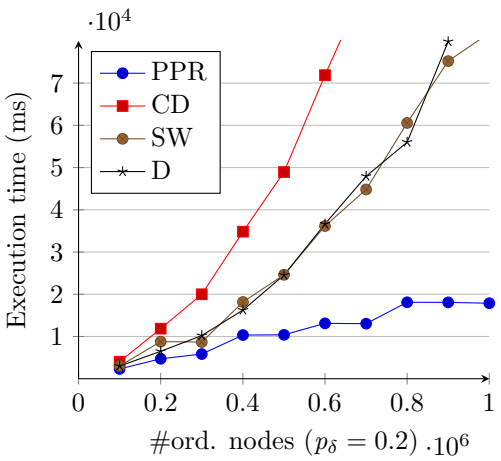


(a) Precomputed chaining application

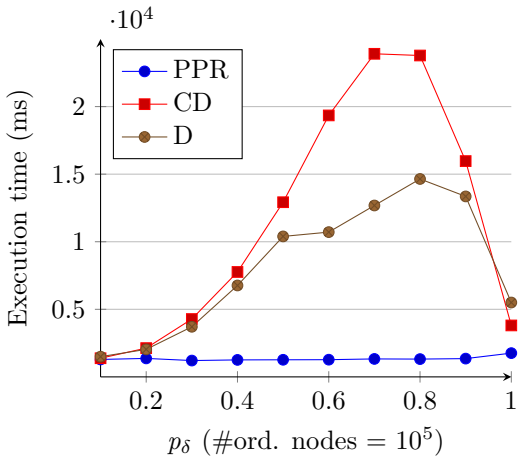
(b) Precomputed chaining application



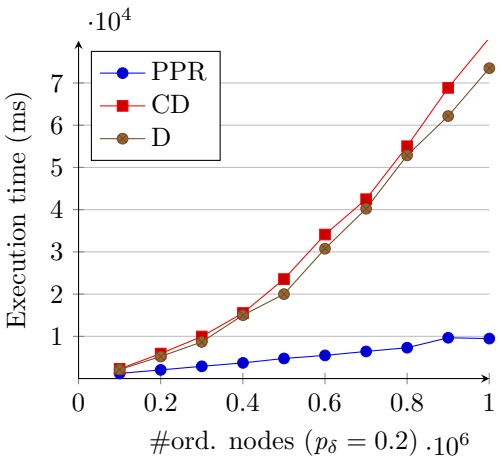
(c) Batch based application



(d) Batch based application



(e) Partition based application



(f) Partition based application

Figure 12.4: Performance of various glue method applications as part of the data mapping

Experimental results for batch based application The behaviour of the BB application is shown in Figure 12.4c for increasing amounts of uncertainty and in Figure 12.4d for increasing data set size. We make the following observations:

- For glue method PPR, we observe constant execution time for increasing amounts of uncertainty and a slight linear increase in execution time for increasing data set size.
- For glue method CD, we observe constant execution time for increasing amounts of uncertainty and exponential behaviour for increasing data set size.
- For glue method D, we observe exponential behaviour for both increasing amounts of uncertainty and increasing data set size.
- For glue method SW, we observe exponential behaviour for $0.1 \leq p_\delta \leq 0.6$ and for increasing data set size. We observe a drop in execution time for $p_\delta > 0.6$.
- The exponential behaviour for increasing data set size of glue methods D and SW is similar. However, for glue method CD, we observe a more rapidly increasing exponential behaviour.

In general, BB requires more phases than PC. For a PC glue method, each node requires one lookup in one of the skeleton path tables while a BB glue method requires n lookups per element where n is the depth of the element table in which that element resides. This difference provides a likely cause for the exponential behaviour of most BB glue process.

However, there are exceptions. Glue method D performs far better in combination with BB than with PC for increasing amounts of uncertainty. The drawback we previously discussed for PC does not apply to BB, because each phase processes possibility nodes of a specific depth. Hence, the evaluation of BB.D does not depend on the distribution of the depth property in element tables. As a consequence, we observe that the BB.D glue process is not affected by the the amount of uncertainty of a data set. For increasing data size, we observe that BB.D performs significantly better than PC.D.

The BB application performs best in combination with PPR for increasing amounts of uncertainty and increasing data set size. Note that PPR has a better performance with PC. The performance of BB.D and BB.PPR are similar for increasing amounts of uncertainty.

Experimental results for partition based application The behaviour of PB is shown in Figure 12.4e for increasing amounts of uncertainty and in Figure 12.4f for increasing data set size. We make the following observations:

- For glue method SW, no performance results are shown, since this method did not finish within the 100 seconds for any of the data sets.
- For glue methods PPR and CD, we observe the same behaviour as for BB. However, both glue methods show a less rapidly increase in execution time for increasing data set size. This indicates a reduction in complexity. For increasing amounts of uncertainty, we also observe an improvement in performance with a constant factor.
- For glue method D, we observe the same behaviour as for glue method CD. However, the performance peak is less high.
- For glue method CD, we observe that the exponential behaviour for increasing data set size increases for PB is less rapidly compared to BB.

The PB application creates complete world set descriptors for a partition of rows in one phase. Since the result of a PB phase is not used in following phases, no intermediate indices have to be created. This differs from BB, where the result of a phase is used as input for the following phase. The time to create intermediate indices is likely to cause the performance difference.

From a database perspective, a PB phase can be described as the concatenation of BB phases with unmaterialized views –views are described in Section 2.6. A concatenation of queries with unmaterialized views allows a database to select a query plan from a wider set of query plans compared to the set of query plans that apply to the evaluation the same concatenation with materialized views at the cost of a much more complex planning phase. As a consequence, the query planner of a URDBMS is more likely to select an inefficient query execution plan if the

planning phase is too complex. In our opinion, the performance of PB.SW and PB.D for increasing amounts of uncertainty suffer from a too complex planning phase.

Sub-conclusions Glue method PPR performs best for all three tested application flavours regardless of the test set. The best performance of PPR is accomplished with PC.

12.3.1.2 Performance study on complete P-XML into URDBMS data mappings

In this section, we present the performance of different P-XML into URDBMS data mappings for increasing amounts of uncertainty and increasing data set size. Our starting point is to select the best performing glue method per application flavour based on the experimental results in Section 12.3.1.1. We deviate from this by selecting glue method CD for the PC application instead of glue method PPR. CD has a similar complexity as PPR for the PC application and allows us to get more diversity in our test results. Otherwise, each test would involve the PPR glue method.

Figures 12.5a and 12.5b present the experimental results for a data mapping that incorporates PC.CD. Figures 12.5c and 12.5d present the experimental results for a data mapping that incorporates BB.PPR. Figures 12.5e and 12.5f present the experimental results for a data mapping that incorporates PB.PPR.

Flesh & skeleton In general, all P-XML into URDBMS data mappings consist of a flesh mapping and a skeleton mapping. In order to evaluate *tg*-queries with a URDBMS, a mapping of the flesh and the skeleton suffice as data mapping. However, in order to evaluate *t*-queries with a URDBMS, a document glue process has to glue flesh and skeleton together. Our flesh mapping is discussed in Section 8.4 and our skeleton mapping is discussed in Sections 8.3 and 11.1. Observe that the execution time of these mappings is the same for all P-XML into URDBMS data mappings that operate on each of the two test sets.

Figures 12.5a, 12.5c and 12.5e show the performance of different data mappings for increasing amounts of uncertainty. Observe that the execution time of f_{fl} is constant. The cause of this is that the set of mapped ordinary nodes is of a constant size for all data sets. In contrast, the behaviour of the skeleton mapping shows quadratic behaviour. As we described in Section 12.2.1, the size of mapped distributional nodes also increases quadratically for increasing amounts of uncertainty.

Figures 12.5b, 12.5d and 12.5f show the performance of different data mappings for increasing data set size. Observe that the query execution time for f_{fl} and for f_{sk} increases linearly with the amount of ordinary nodes in the data set. Since the number of distributional nodes also increases linearly with the data set size, we identify a linear relation between f_{sk} execution time and number of distributional nodes in the data set. More specifically, f_{fl} has an average throughput of $6.6 \cdot 10^4$ ordinary nodes per second and f_{sk} has an average throughput of $7.0 \cdot 10^3$ possibility nodes per second.

Skeleton path tables Figures 12.5a and 12.5b show the stacked execution times for data mappings that incorporate a PC glue process. As we explained in Section 10.1, PC requires skeleton tables to be created prior to the glue process. We create skeleton tables with a slightly altered glue process that is either BB or PB. We select BB.PPR to build the skeleton path tables in our experiments since this glue process is the most efficient according to Section 12.3.1.1. Experimental results show that the performance of skeleton path tables is constant for $0.1 \leq p_\delta 0.6$ and for increasing data set size. We observe a linear increasing behaviour for $p_\delta > 0.6$.

Glue process It follows from our experimental results that PB.PPR is the most efficient glue process as part of a data mapping. More specifically, PB.PPR has an average throughput of $1.0 \cdot 10^5$ ordinary nodes per second. The performance of other DO glue processes are discussed in Section 12.3.1.1.

Sub-conclusions Based on the observations made in this section, we present the following conclusions:

- The execution time of the flesh mapping increases linearly with the number of ordinary nodes involved in the mapping. The execution time of the skeleton mapping increases linearly with the number of distributional nodes involved in the mapping.
- The performance advantage of a data mapping that incorporates a PC glue process does not outweigh the execution costs to create skeleton path tables (skeleton path tables have to be created in order to use PC).
- A DO glue process is performed most efficiently with the PB.PPR glue process regardless of the test set. Of course, a DO glue process may be omitted to improve performance of a P-XML into URDBMS data mapping. As a result, *tg*-queries have to be evaluated instead of *t*-queries.

12.3.2 Experimental results for XPath evaluation by a URDBMS

In this section, we present the experimental results for XPath evaluation by a URDBMS. In particular, we focus on the performance difference between *t*-query evaluation and *tg*-query evaluation. The main advantage of *tg*-query evaluation is that a P-XML into URDBMS data mapping does not include a DO glue process. A performance comparison between *t*-query evaluation and *tg*-query evaluation has to point out if a performance advantage of the data mapping outweighs a possible performance loss with *tg*-query evaluation instead of *t*-query evaluation.

We use *t*-query evaluation as baseline in order to express performance of *tg*-query evaluation in terms of normalized speedup. We use normalized speedup as measure to express how many times the evaluation of a specific *tg*-query kind is faster than the evaluation of *t*-queries. We present performance results of various kinds of *tg*-queries. Each kind is identified with the unique glue process it incorporates.

12.3.2.1 Experimental results for query evaluation on increasing data set size

Figures 12.6a and 12.6b show a performance comparison between *tg*-query and *t*-query. We use the performance of *t*-query evaluation as baseline to calculate the normalized speedup of *tg*-query evaluation. Note that normalized speedup is calculated for the total normalized query execution costs of a query category. A higher normalized speedup indicates a better performing and more robust approach against document size fluctuations.

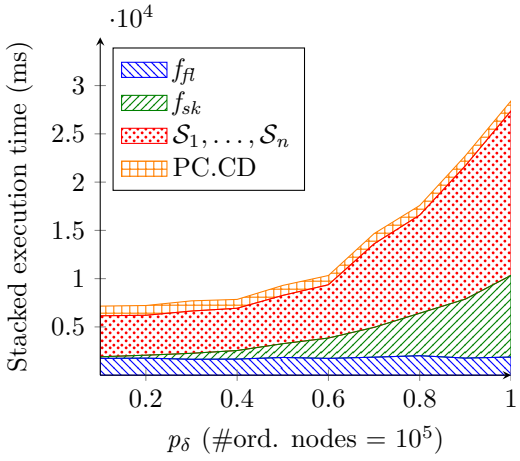
To get a better understanding of normalized speedup, we give a small illustration. Figure 12.6b shows a normalized speedup of 3 for query category 3 with glue process PC.CD (mv). A normalized speedup of 3 indicates that for a random selected query in category 3 and a random selected data set in the test set for increasing data set size, the actual speedup is on average 3.

We make the following observations:

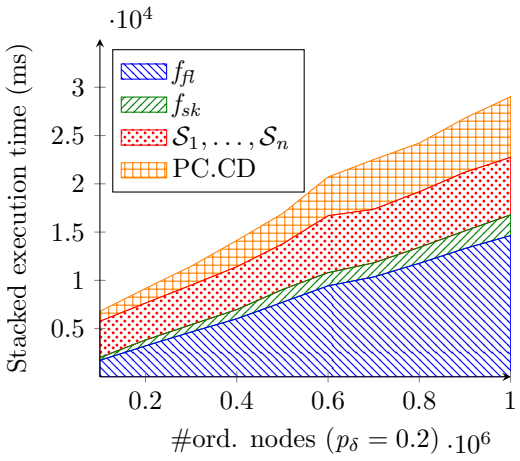
1. Figure 12.6a: we observe that glue processes that use a materialized view outperform glue processes that use an unmaterialized view for query categories 1,2,4,5. This indicates that the query planner selected inefficient query evaluation plans. For future observations, we assume the performance of a glue process to be denoted by the highest of the (v) or (mv) column.

2. Figure 12.6a: we observe that *tg*-queries that incorporate PC.PPR perform best for categories 1 (complex predicate), 2 (foll/prec sibling) and 5 (simple predicate).

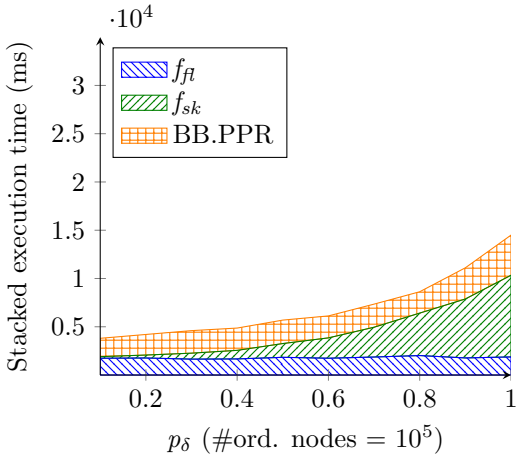
3. Figure 12.6a: we observe that *tg*-queries that incorporate PC.CD_*TI* perform best for categories 3 (anc/desc) and 4 (par/child).



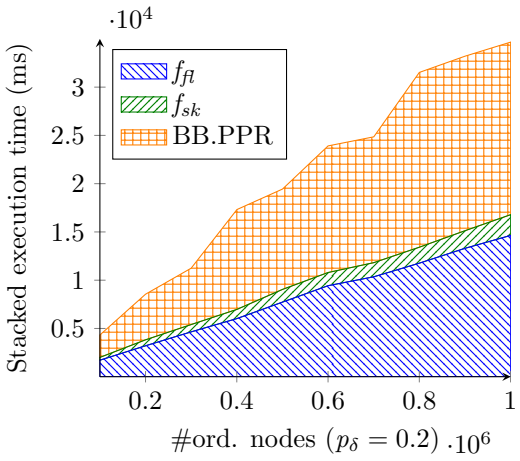
(a) Stacked area graph for PC.CD



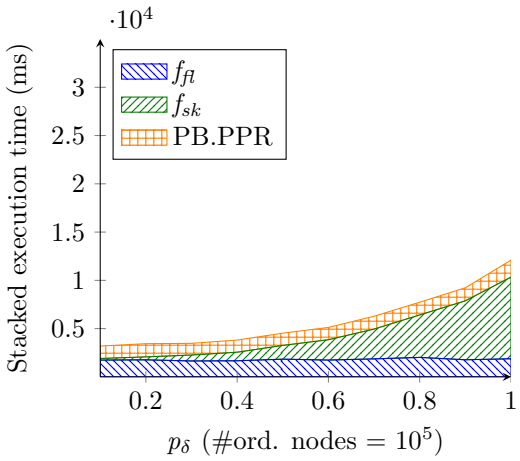
(b) Stacked area graph for PC.CD



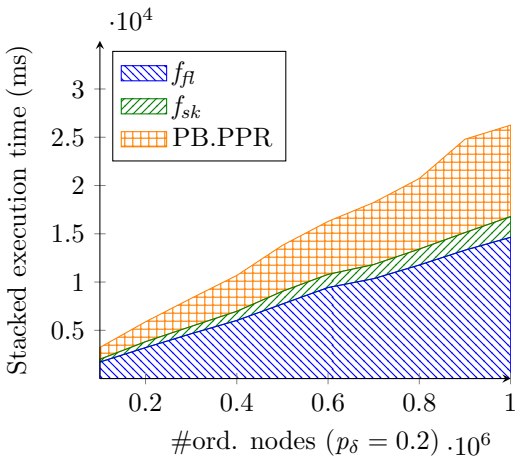
(c) Stacked area graph for BB.PPR



(d) Stacked area graph for BB.PPR



(e) Stacked area graph for PB.PPR



(f) Stacked area graph for PB.PPR

Figure 12.5: Data Mapping including Document Gluing

4. Figure 12.6b: we observe that *tg*-queries have a better performance in general than *t*-queries for category 1 (complex predicate), category 2 (anc/desc) and category 4 (anc/desc). Furthermore, *tg*-queries that incorporate a PC glue process also outperform *t*-queries for category 5 (simple predicate).

5. Figure 12.6b: we observe that *t*-queries have a better performance in general than *tg*-queries for category 3 (anc/desc), accept for *tg*-queries that incorporate PC.CD_*TI*.

6. Figure 12.6b: we observe that *tg*-queries that incorporate PC glue process have a better performance than *tg*-queries that incorporate a BB glue process for categories 1,4 and 5.

7. Figure 12.6c: we observe the average query execution time for queries of a certain category evaluated with a certain glue process. These results seem promising:

   (a) Category 1: query evaluation time with PC.PPR (v) is 2.4s on average;
   (b) Category 2: query evaluation time with PC.PPR (v) is 0.24s on average;
   (c) Category 3: query evaluation time with PC.CD_*TI* (v) is 0.20s on average;
   (d) Category 4: query evaluation time with PC.CD_*TI* (mv) is 0.8ms on average;
   (e) Category 5: query evaluation time with *t*-queries is 17ms on average.

More detailed experimental results are found in Appendix C.4.1. These results show the effects of increasing data set size on the normalized speedup per glue method. We observe that the normalized speedup of most *tg*-queries is constant. This indicates that the complexity of evaluating *tg*-queries and *t*-queries is the same. One notable exception is the normalized speedup of glue process PC.CD_*TI* for category 3. We observe a linear increase in normalized speedup that indicates a reduction in complexity.

Sub-conclusions Glue process PC.PPR performs best in most categories. However, in scenarios that profit from inlining, glue processes that allow for a loosening of the inline rules perform significantly better than other glue processes.

12.3.2.2 Experimental results for query evaluation on increasing amount of uncertainty

Figures 12.7b and 12.7a show the normalized speedup for *tg*-query evaluation compared to *t*-query evaluation for varying amounts of uncertainty. Note that we calculate the speedup for the total normalized query execution costs. A higher normalized speedup indicates a better performing and more robust approach against varying amounts of uncertainty.

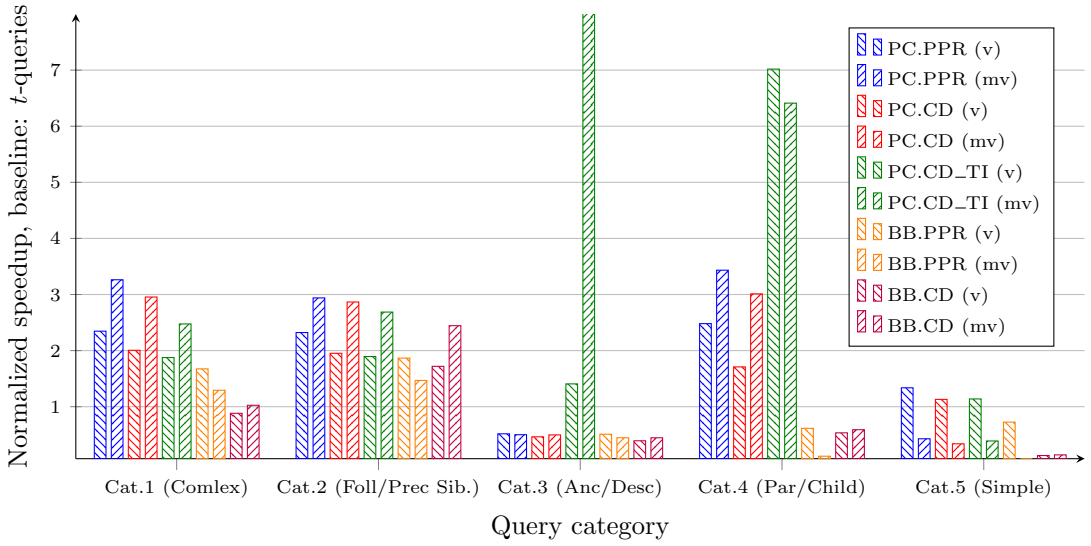
We make the following observations:

1. Figure 12.7a: we observe that glue processes that use a materialized view outperform glue processes that use an unmaterialized view for query categories 1,3,4,5. This indicates that the query planner selected inefficient query evaluation plans. For future observations, we assume the performance of a glue process to be denoted by the highest of the (v) or (mv) column.

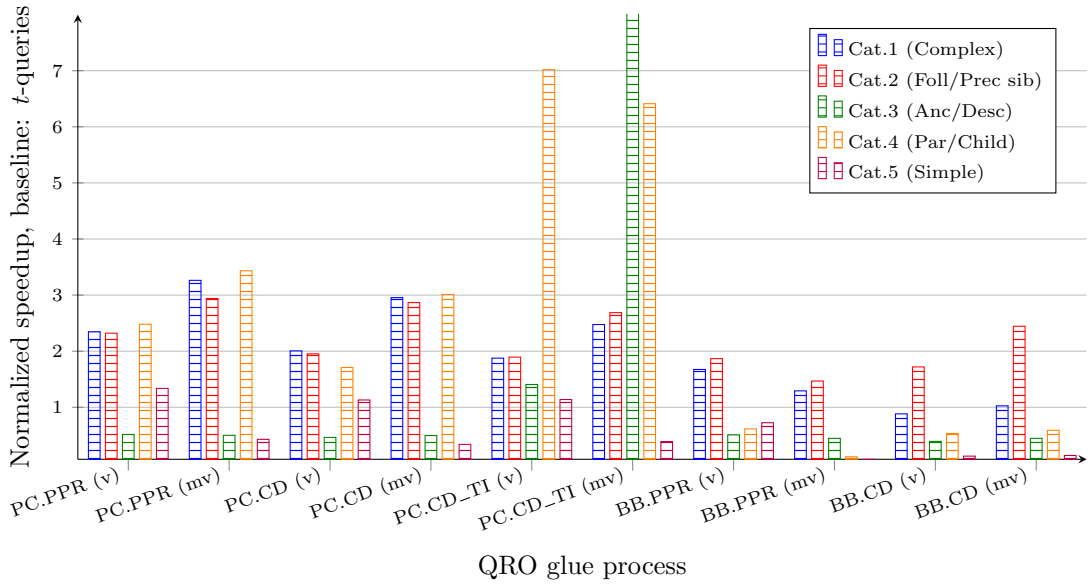
2. Figure 12.7a: we observe that *tg*-queries that incorporate PC.PPR perform best for categories 1 (complex predicate) and 2 (foll/prec sibling).

3. Figure 12.7a: we observe that *tg*-queries that incorporate PC.CD_*TI* perform best for categories 3 (anc/desc) and 4 (par/child). In both categories, we observe a large performance difference with PC.CD. We attribute this performance improvement to a loosening of the inline rules.

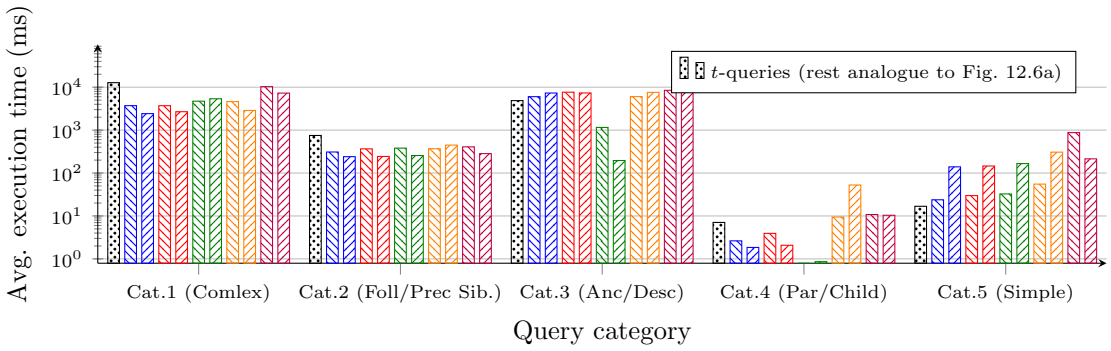
4. Figure 12.7a: we observe that *t*-queries and *tg*-queries that incorporate PC.PPR perform best for category 5 (simple predicate).



(a) Total over varying amounts of data set size {100.000,200.000,. . . ,1000.000}



(b) Total over varying amounts of data set size {100.000,200.000,. . . ,1000.000}



(c) Average query execution time per query –categorized per query category

Figure 12.6: Query execution results for test set of increasing data set size

5. Figure 12.7a: we observe a similar performance of all approaches to evaluate queries in category 2 (foll/prec sibling) and category 3 (anc/desc).

6. Figure 12.7b: we observe that *tg*-queries have a better performance than *t*-queries for category 1 (complex predicate), category 2 (foll/prec sibling) and category 3 (anc/desc).

7. Figure 12.7b: we observe that *tg*-queries that incorporate a PC glue process have a better performance than *tg*-queries that incorporate a BB glue process for all categories. However, the performance difference for categories 2 and 3 are negligible.

8. Figure 12.7c: we observe the average query execution time for queries of a certain category evaluated with a certain glue process. These results seem promising:

   (a) Category 1: query evaluation time with PC.PPR (v) is 44ms on average;

   (b) Category 2: query evaluation time with PC.PPR (mv) is 8ms on average;

   (c) Category 3: query evaluation time with PC.CD_*TI* (mv) is 0.2ms on average;

   (d) Category 4: query evaluation time with PC.CD_*TI* (v) is 0.8ms on average;

   (e) Category 5: query evaluation time with *t*-queries is 7.8ms on average.

More detailed experimental results are found in Appendix C.4.2. These results show the effects of increasing amounts of uncertainty on the normalized speedup per glue method. We observe that normalized speedup of most *tg*-queries is constant. This indicates that the complexity of evaluating *tg*-query and *t*-query is the same.

Sub-conclusions Glue process PC.PPR performs best in most categories. However, in scenarios that profit from inlining, glue processes that allow for a loosening of the inline rules perform significantly better than other glue processes.

12.3.2.3 Experimental results for query evaluation on a real world uncertain test set

Figures 12.8a and 12.8b show the normalized speedup for *tg*-query evaluation compared to *t*-query evaluation per query category and per glue process. A higher speedup indicates a more robust approach against a varying movie title threshold. This threshold is the factor that is varied for the real world uncertain test set. More details about this test set are found in Section 12.2.1.

Due to time limitations, we were unable evaluate XPath queries with *tg*-queries that use materialized views. As a consequence, we cannot determine if the query planner selected relative efficient query plans.

- Figure 12.8b: in correspondence with the results of Figure 12.8b, we observe that PC glue processes have a similar performance than *t*-queries.

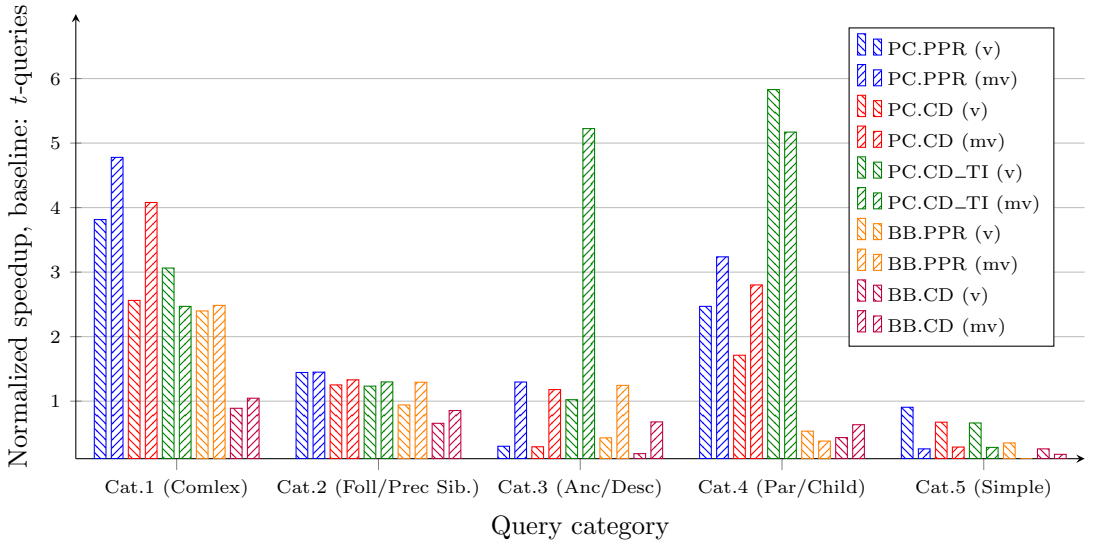
- Figure 12.8b: unlike Figure 12.7a, we observe that the performance difference between PC.CD_TI and PC.CD is minimal. Furthermore, for category 3, we observe that *t*-queries perform better than all kinds of *tg*-queries. For category 4, we observe that PC.CD_*TI* performs better than PC.CD, but is outperformed by PC.PPR.

- Figure 12.8b: unlike Figure 12.8b, we observe that all kinds of *tg*-queries have a poorer performance than *t*-queries for category 3, including PC.CD_*TI*.

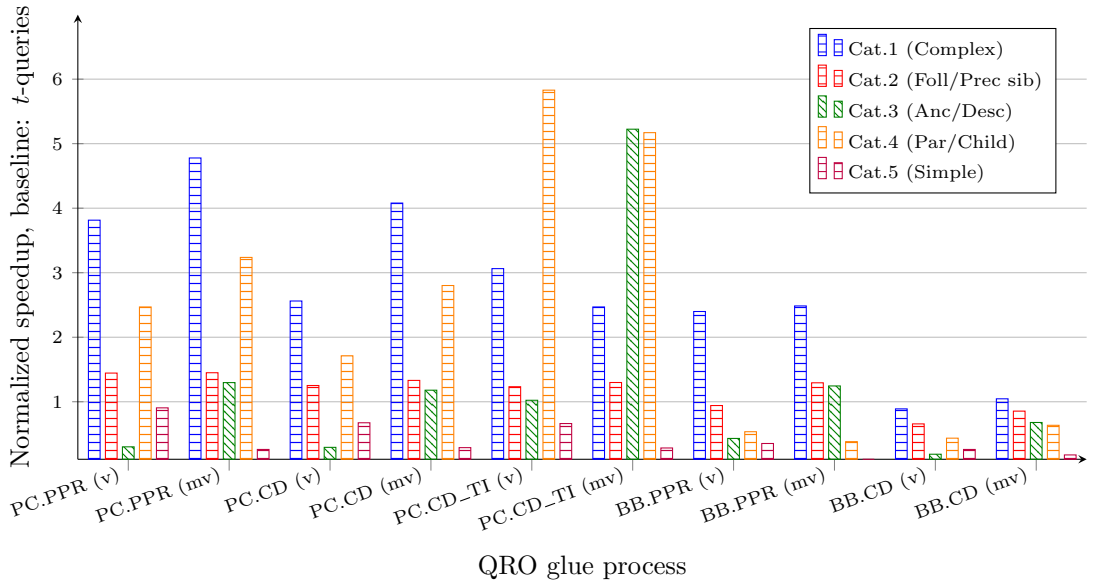
- Figure 12.8b: like Figure 12.7a, we observe *t*-queries to perform best for category 5.

- Figure 12.8b: like Figure 12.7a, we observe that *tg*-queries that incorporate PC.PPR perform best for categories 1 and 5.

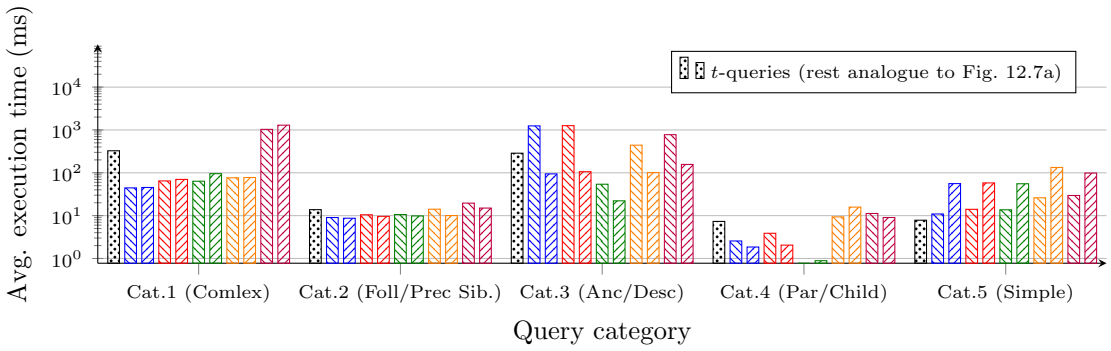
- In Figure 12.8c, the average query execution time for query evaluation on a real world uncertain test set are shown. We highlight the best performing approach per query category:



(a) Total over varying generated amounts of uncertainty (p_δ varied over {0.1,0.2,...,1.0})



(b) Total over varying generated amounts of uncertainty (p_δ varied over {0.1,0.2,...,1.0})



(c) Average query execution time per query –categorized per query category

Figure 12.7: Query execution results for test set of increasing amount of uncertainty

1. Category 1: query evaluation with PC.PPR (v) is 20ms on average;
2. Category 2: query evaluation with PC.CD_*TI* (v) is 5.8ms on average;
3. Category 3: query evaluation with *t*-queries is 3.7ms on average;
4. Category 4: query evaluation with PC.PPR (v) is 11ms on average;
5. Category 5: query evaluation with PC.PPR (v) is 84ms on average;

Query results of a few milliseconds per query are highly sensitive for measurement errors. However, the experimental results in Appendix C.4.3 show perfect constant normalized speedup for all categories. This is an indication that our experimental results did not suffer from measurement errors.

Sub-conclusions Based on the previous observations on an uncertain real world test set, we conclude that a loosening of the inline rules does not contribute to a significant performance advantage, this in contrast to previous experiments. For XPath expressions with complex predicates, observations confirm that *tg*-queries perform significantly better than *t*-queries. For categories 3 and 5, *t*-queries perform best. *tg*-queries perform best with a PC.PPR glue process.

12.3.3 Conclusions

We make a distinction between conclusions that have emerged from experiments on data mapping performance and experiments on query evaluation performance.

Data mapping – flesh & skeleton mapping Performance of f_{fl} scales linearly with the number of ordinary nodes that are mapped. Analogously, the performance of f_{sk} increases linearly with the number of possibility nodes. Additionally, the execution time to create skeleton path tables is constant for $0.1 \leq p_\delta \leq 0.6$. For larger values of p_δ, execution time increases linearly.

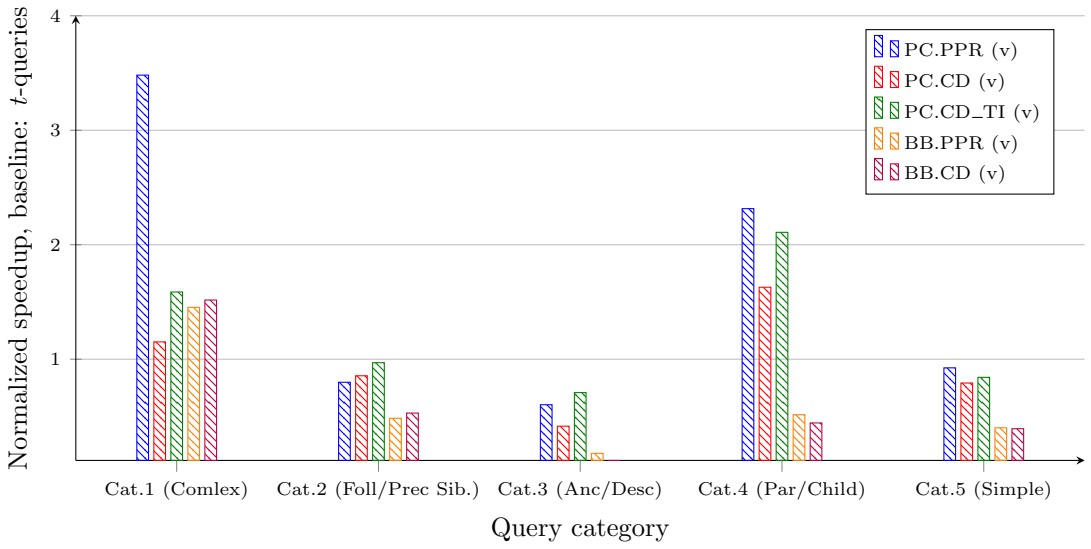
Data mapping – best DO glue process The PB.PPR glue process is the best candidate to incorporate in a data mapping. The execution time of PB.PPR increases linearly with the number of ordinary nodes. The number of possibility nodes in a data set does not influence the execution time of PB.PPR.

Query evaluation – unmaterialized views vs. materialized views For the test set of increasing data set size and the test set of increasing amounts of uncertainty, performance results indicate that materialized views perform better than unmaterialized views in many scenarios. For category 1 (complex predicate) and category 3 (par/child), this was more rule than exception. We interpret this behaviour as an indication that the query planning process was too complex for the query planner.

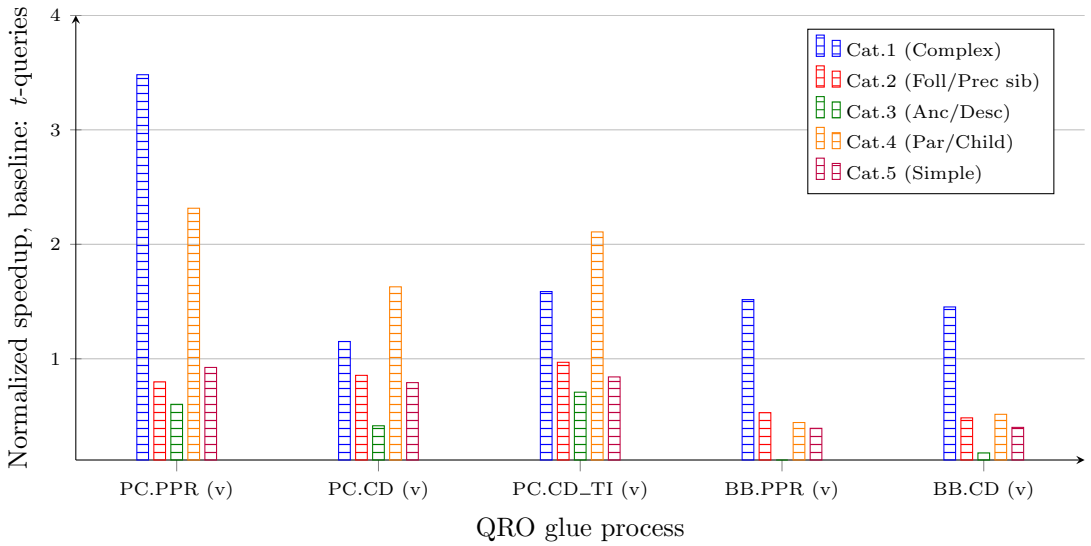
Query evaluation – same complexity In general, the normalized speedup of most *tg*-queries compared to *t*-queries is constant. This indicates that the complexity of the evaluation of *tg*-queries and *t*-queries is the same.

Query evaluation – performance of *tg*-queries compared to *t*-queries In general, *tg*-queries perform best with an incorporated PC glue process. Especially, *tg*-queries that incorporate PC.PPR outperform *t*-queries in most scenarios. A short summary of the performance results:

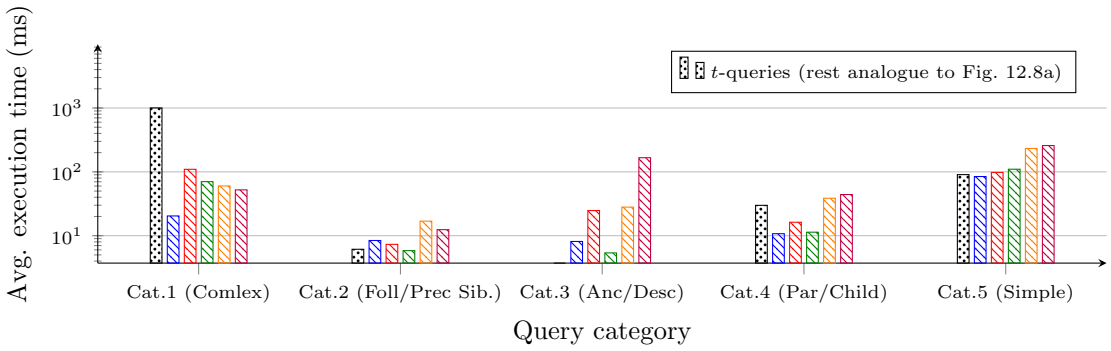
- For varying data set sizes, *tg*-queries that incorporate PC.PPR perform better or similar to *t*-queries in categories 1 (complex predicate), 2 (foll/prec sibling), 4 (par/child), 5 (simple predicate). For category 3 (anc/desc), *t*-queries have a better performance.
- For varying amounts of uncertainty, *tg*-queries that incorporate PC.PPR perform in all categories better or similar to *t*-queries.



(a) Total over varying movie title thresholds



(b) Total over varying movie title thresholds



(c) Average query execution time per query –categorized per query category

Figure 12.8: Query execution results for real world uncertain test set

- For the real world uncertain test set, *tg*-queries that incorporate PC.PPR perform better or similar to *t*-queries in categories 1 (complex predicate), 2 (foll/prec sibling), 4 (par/child), 5 (simple predicate). For category 3 (anc/desc), *t*-queries have a better performance.

Note that skeleton path tables have to be created as part of a data mapping in order to use PC. As a consequence, a data mapping is less efficient.

Query evaluation – advantages of inlining Our experimental results show that queries from category 3 (anc/desc) and category 4 (par/child) profit from a loosening in the inlining rules. More specifically, we managed to get a linear reduction in complexity for increasing data set sizes and an constant increase of 8 for normalized speedup.

Unfortunately, experimental results on the real world uncertain test set show that in practice, the performance advantage as a result of inlining is minimal. The reason for this is rather trivial. The traditional inlining rules –described in Section 2.3.1– apply to the flesh of a p-document instead of the individual possible worlds represented by that p-document. Our approach to add uncertainty to an XML-document basically makes edges between nodes uncertain. Hence, the multiplicity between element types is not affected. However, in practice, nodes of a p-document are uncertain. As a consequence, for element types N and N' that have a multiplicity of 1..1 in each possible document represented by a p-document have a multiplicity of $1..*$ in the flesh of that p-document in case one occurrence of N' is uncertain. Hence, the traditional inlining rules prohibit the inlining of N' with N.

We conclude that performance benefits of inlining is minimal in practice.

Chapter 13

Related Work

This work is a continuation of previous work [48] where we designed a P-XML into URDBMS data mapping based on the XPath Accelerator (XA) approach of Grust et al. [20] combined with the PB.D glue process. We used this data mapping to perform an XPath performance study on MayBMS. Results indicated that query evaluation was inefficient. In this thesis, we managed to resolve this issue with a change of XA element encoding and a change of XML into RDBMS data mapping from XA to ASI[*XA*].

The work of Van Keulen et al. [50] gave rise to investigate XPath processing with a URDBMS. They propose two approaches to build an XPath processor for P-XML data. The first approach is to instruct a URDBMS to cope with XML. The second approach is to instruct an XML RDBMS to cope with uncertainty. They implemented the second approach. No results on scalability are reported.

Besides our work, the work of Hollander et al. [26] also investigated the possibility to transform a URDBMS to an XPath processor. They propose two different P-XML into URDBMS data mappings. Their first mapping extends the XA approach and their second mapping extends the Shared Inlining (SI) approach, both in combination with 'Trio' –a URDBMS. In summary, their approach consist of three steps: (1) create an event table that represents the skeleton of a p-document, (2) map the flesh of a p-document to relation tables (3) create uncertain relation tables per element type by joining the flesh with the skeleton using an inheritance driven glue method –discussed in Section 10.2. The event table that Hollander uses in the first step is similar to the result of the flesh mapping introduced in this research

The performance study of Hollander et al. [26] shows that their SI approach is more efficient than their XA approach for query evaluation. Unfortunately, Trio was unable to import large documents. As a consequence, their performance study is of a different order of magnitude than the performance study of this work.

Most work on query evaluation on P-XML data is of Kimelfeld et al. [34, 32, 33]. They propose a variety of algorithms for which they give a complexity analysis. Their work is not described with the same detailed level as our work. Also, they have not shown that their algorithms scale up to practice. In contrast, our work has not yet achieved a complexity analysis. Provisionally, it remains the question how the work of Kimelfeld et al. fits with our work.

Nierman et al. [41] have implemented a query evaluation mechanism for the probabilistic XML model member of the $\mathsf{PrXML}^{\{\mathsf{exp}\}}$ family. The work of Nierman et al. is not comparable with our approach, since we investigated a P-XML model of a different probabilistic XML family.

Chapter 14

Conclusions & Future Work

14.1 Summary

In this thesis, we identified the following problem:

> *There does not exists an efficient query evaluation mechanism for P-XML that can be used in practice.*
> — Section 1.2

In order to resolve this problem, we take the approach to instruct a uncertain relational database management system (URDBMS) to cope with P-XML. We accomplish this with a specification and design of P-XML into URDBMS database mapping (f, g) where f is a data mapping that maps p-documents to U-Relations and g is a query mapping that maps XPath to SQL queries.

A p-document constructed of flesh and skeleton Document instances of P-XML are built up from ordinary nodes and distributional nodes. We introduce the terminology *flesh* to refer to all ordinary nodes in a p-document and *skeleton* to refer to all distributional nodes in a p-document. The flesh defines the content of a p-document and the skeleton defines the uncertainty distribution of a p-document as a set of choices.

First design of (f, g) Our first design constructs f as (1) a flesh mapping (f_{fl}) to map tree-structured content to table-structured content and (2) a skeleton mapping (f_{sk}) to map a tree-structured uncertainty distribution to a table-structured uncertainty distribution. Additionally, we use (3) a glue process to merge the result of f_{fl} and f_{sk}. The result of such glue process is a set of U-Relations that represents the same set of possible worlds as the original p-document. We refer to a glue process that is part of f as document oriented gluing (DO).

In correspondence with f, we construct g as an ordinary XPath to SQL mapping. According to the possible worlds semantics –described in Section 2.4.1–, a traditional query evaluated on a set of possible worlds results in a set of possible answers, each provided by one of the possible worlds. Therefore, the result of a traditional XPath query evaluated on a tree-structured representation of a set of possible worlds is similar to the result of a traditional SQL query –derived from g– evaluated on a table-structured representation of possible worlds –derived from f. We refer to queries derived from an ordinary XPath to SQL mapping as t-queries.

Second design of (f, g) A P-XML into URDBMS data mapping is not obliged to incorporate a glue process. Our second design constructs f as f_{fl} and f_{sk}. Query mapping g is altered such that a glue process is incorporated in the query evaluation process. We refer to a glue process that is part of a query evaluation process as query result oriented gluing (QRO). Queries that include a glue process are referred to as tg-queries.

Experimental validation We conducted an extensive experimental evaluation of both designs on synthetically generated data sets and real-world data sets.

Experimental results on the mapping of P-XML data indicate that the execution time of f_{fl} increases linearly for increasing amounts of ordinary nodes and the execution time of f_{sk} increases linearly for increasing amounts of distributional nodes. For the first design, the glue process PB.PPR is most efficient as additional glue process. The execution time of this glue process increases linearly with the number of ordinary nodes and is unaffected by the number of distributional nodes.

Experimental results on the evaluation of XPath queries indicate that tg-queries and t-queries have the same complexity. In most scenarios, XPath queries evaluated as tg-queries that incorporate PC.PPR as glue process are most efficient.

14.2 Evaluation of research questions

In the introduction of this thesis, we specified a set of research questions. We answer these questions one by one.

> RQ1: ***How do we correctly evaluate XPath queries on P-XML as SQL queries on a URDBMS?***

Our goal is to specify (f, g) where f is a P-XML into URDBMS data mapping and g is a XPath to SQL query mapping such that the same set of possible worlds is represented under f and the same question is specified under g. Hence, for each XPath query q and p-document pd, we obtain the answer of q evaluated on pd as $g(q)$ evaluated on $f(pd)$.

We specify (f, g) as a sequential composition of four other mappings. This approach is illustrated in Figure 14.1. A double headed arrow denotes a database mapping constructed as a data mapping –a mapping from one data representation to another– and a query mapping –a mapping from one query language to another. For (f_i, g_i) varied over each illustrated database mapping, we show that the same set of possible worlds is represented under f_i and the same question is specified under g_i. It follows that the same set of possible worlds is represented under the sequential composition of $f_1, \ldots, f_4$ and the same question is specified under the sequential composition of $g_1, \ldots, g_4$.

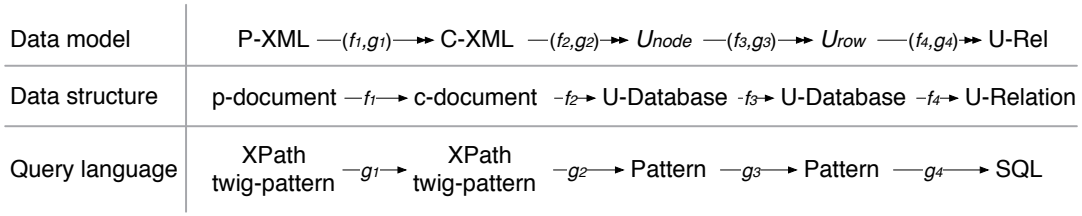


Figure 14.1: A P-XML into URDBMS mapping as a sequential composition

Part II (Specification) discusses the abovementioned approach in more detail. We present our advancing understanding of each of the mappings that form the basis of our specification for (f, g).

The general principle behind our approach to construct (f, g) is to represent different uncertain data models –in this case, P-XML and U-Rel– as $\mathcal{U}$ and define a mapping between the two instantiations of $\mathcal{U}$. Due to the genericity of $\mathcal{U}$, this formalisms can be used to bridge the gap between other uncertain data models as well; construct one complex database mapping as a sequence of less complex database mappings that use $\mathcal{U}$ as intermediate data model.

> RQ2: ***How do we efficiently map P-XML data into a URDBMS?***

In this thesis, we propose two designs for (f, g). The first design constructs f as a glue process that uses f_{fl} and f_{sk}. The second design constructs f as f_{fl} and f_{sk} without a glue process. We describe optimizations for f_{fl}, f_{sk} and an optional glue processes.

Optimizations for f_{fl} Function f_{fl} maps ordinary nodes to tables. We define f_{fl} as ASI[XA]; an XML into RDBMS mapping derived from the relational data structure and inline principle of Shared Inlining (SI) and the element encoding of XPath Accelerator (XA). ASI[XA] enables a depth reduction optimization that led to a more compact representation of possible worlds and more efficient query evaluation. This advantage is in effect orthogonal to the selection to use a DO glue process –a glue process as part of f– or a QRO glue process –a glue process as part of the query evaluation process.

Benchmark results show that the performance of f_{fl} scales linearly with the number of ordinary nodes mapped. More specifically, f_{fl} has an average throughput of $6.6 \cdot 10^4$ ordinary nodes per second.

Optimizations for f_{sk} Function f_{sk} represents distributional nodes as RVAs in a URDBMS. This enables a tree-structured uncertainty distribution to be represented as a table-structured uncertainty distribution. RVAs are created with the repair-key statement (f_{rk}) of MayBMS. In Section 11.1, we identified a performance problem for this statement that we managed to solve with the multi-union approach.

Benchmark results indicate the performance of f_{sk} scales linearly with the number of possibility nodes. More specifically, f_{sk} has an average throughput of $7.0 \cdot 10^3$ possibility nodes per second.

Optimizations for (optional) document oriented glue process A DO glue process is part of our first design of (f, g). This process merges the results of f_{fl} and f_{sk} such that its result represents a set of possible worlds. A DO glue process is part of a data mapping and is optional. Either a glue process is incorporated as part of the data mapping –first design– or as part of the query evaluation process –second design.

A glue process is defined as a glue method application and a glue method. We introduce three kinds of applications and four kinds of glue methods. Each combination of a glue method and an application results in a valid glue process.

Benchmark results indicate the PB.PPR glue process to be the best pick as DO glue process. Execution time of PB.PPR increases linearly with the number of ordinary nodes and is unaffected by the number of distributional nodes. More specifically, PB.PPR has an average throughput of $1.0 \cdot 10^5$ ordinary nodes per second.

RQ3: ***How do we efficiently evaluate XPath queries on P-XML data on a URDBMS?***

Our first design evaluates XPath queries on P-XML data as t-queries on the result of a DO glue process. t-queries are SQL queries derived from an ordinary XPath to SQL mapping.

Our second design evaluates XPath queries on P-XML as tg-queries on the results of f_{fl} and f_{sk}. tg-queries are SQL queries derived from an ordinary XPath to SQL mapping extended with a glue process. Our second design enables a number of optimizations. One optimization uses tree-information from XPath expressions in order to reduce the number of iterations of a QRO glue process. Another optimization increases the number scenarios that profit from the inlining principle. These optimizations make it possible to perform a glue process as part of query evaluation without ruining query evaluation performance.

Benchmark results indicate tg-queries that use PC.PPR as glue process to perform best in most scenarios. The only exception are XPath queries that extensively use the parent/child axis. This type of XPath queries are more efficiently evaluated as t-queries.

14.3 Research goal achievement

We have achieved our research goal with the answering of previously mentioned research questions. We have built an XPath processor for P-XML data that is scaled up to practice. This XPath processor is constructed as a P-XML into URDBMS database mapping on top of MayBMS. Additionally, we sketch an approach to show that the answer of an XPath query on P-XML data evaluated with each of our two designs is correct.

14.4 Future work

Correctness In part II (Specification) of this thesis, we provide all necessary definitions in order to prove correctness of our approach to transform a URDBMS into an XPath processor for P-XML data. We give our advancing understanding of each of the mappings that form the basis of our specification for a correct P-XML into URDBMS mapping. We leave the actual proof for future work.

P-XML expressiveness We investigate XPath evaluation for the P-XML data model of Van Keulen et al. [50]. As we stated in Section 2.5.1, this data model is member of the $\mathsf{PrXML}^{\{\mathsf{ind},\mathsf{mux}\}}$ family [33]. We believe that our research is also applicable to P-XML data models member of the more expressive P-XML family $\mathsf{PrXML}^{\{\mathsf{cie}\}}$. A sketch of such application is found in Appendix D.

Comparison with related work In the related work chapter, we mentioned the work of Kimelfeld et al. [34, 32, 33]. In order to compare their algorithms with our approach, a complexity analysis of this work should be made.

Extend to another URDBMS Developers of the Monte Carlo Database System (MCDB) have shown an interest towards probabilistic XML [43]. Future research should indicate how well our generic principles can be combined with MCDB.

Overhead of uncertainty management Multiple contributions aim to reduce the depth of a query in order to improve performance. The largest reduction in query depth is accomplished for *tg*-queries. As a consequence, we managed to get a better overall performance for *tg*-queries than *t*-queries while *tg*-queries are burdened with a glue process. We assume that there is a relation between query depth and performance overhead, however, additional testing should verify the existence of such relation. In more general, we are interested in the performance overhead of uncertainty management.

Comparison between ASI[*XA*] and XA In previous work [48], we constructed a P-XML into URDBMS data mapping on top of the XA approach. In this thesis, we construct a similar data mapping on top of ASI[*XA*], a new XML into RDBMS mapping. We are interested if ASI[*XA*] has an overall better performance on top of a typical RDBMS than XA.

Creating skeleton path tables We identified the PC.PPR glue process to have the best overall performance for XPath evaluation with *tg*-queries. In order to use PC, skeleton path tables have to be created. We selected the best performing DO glue process to achieve this. It is uncertain if the best performing DO glue process performs best for the creation process of skeleton path tables. Additional experiments should indicate if BB.PPR is the best candidate to create skeleton path tables.

Optimizing the repair-key statement The repair-key statement (f_{rk}) of MayBMS is a required component of our P-XML into URDBMS mapping. In Section 11.1, we identified a performance issue with f_{rk}. We managed to solve this issue with the multi-union approach. As an alternative for the multi-union approach, it is possible to internally rewrite f_{rk} as an extended GROUP-BY expression. Due to the similarities between the well-optimized GROUP-BY expression and f_{rk}, we have reason to believe that the evaluation of f_{rk} can even more efficient than we have accomplished so far.

Bibliography

[1] *PostgreSQL Manual*. http://www.postgresql.org/docs/8.3.

[2] Serge Abiteboul, Benny Kimelfeld, Yehoshua Sagiv, and Pierre Senellart. On the expressiveness of probabilistic xml models. *VLDB J.*, 18(5):1041–1064, 2009.

[3] Serge Abiteboul and Pierre Senellart. Querying and updating probabilistic information in xml. In Yannis Ioannidis, Marc Scholl, Joachim Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust, and Christian Boehm, editors, *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 1059–1068. Springer Berlin / Heidelberg, 2006.

[4] Periklis Andritsos, Ariel Fuxman, and Renee J. Miller. Clean answers over dirty databases: A probabilistic approach. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 30–, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 983–992, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Lyublena Antova, Christoph Koch, and Dan Olteanu. Query language support for incomplete information in the maybms system. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1422–1425. VLDB Endowment, 2007.

[7] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: databases with uncertainty and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 953–964. VLDB Endowment, 2006.

[8] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérome Siméon. *XML Path Language (XPath) 2.0 World Wide Web Consortium Candidate Recommendation*, September 2005.

[9] Jihad Boulos, Nilesh Dalvi, Bhushan Mandhani, Shobhit Mathur, Chris Re, and Dan Suciu. Mystiq: a system for finding more answers by using probabilities. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 891–893, New York, NY, USA, 2005. ACM.

[10] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 310–321, New York, NY, USA, 2002. ACM.

[11] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 455–466, New York, NY, USA, 2005. ACM.

[12] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends databases*, 1(4):379–474, April 2009.

[13] Reynold Cheng. Querying and cleaning uncertain data. In Kurt Rothermel, Dieter Fritsch, Wolfgang Blochinger, and Frank Drr, editors, *Quality of Context*, volume 5786 of *Lecture Notes in Computer Science*, pages 41–52. Springer Berlin / Heidelberg, 2009.

[14] Reynold Cheng, Sarvjeet Singh, and Sunil Prabhakar. U-dbms: a database system for managing constantly-evolving data. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1271–1274. VLDB Endowment, 2005.

[15] Sara Cohen and Benny Kimelfeld. Querying parse trees of stochastic context-free grammars. In *ICDT*, pages 62–75, 2010.

[16] Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52:86–94, July 2009.

[17] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, October 2007.

[18] A. Dekhtyar, J. Goldsmith, and S.R. Hawkes. Semistructured probabilistic databases. In *Scientific and Statistical Database Management, 2001. SSDBM 2001. Proceedings. Thirteenth International Conference on*, pages 36 –45, 2001.

[19] Torsten Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 109–120, New York, NY, USA, 2002. ACM.

[20] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29:91–131, March 2004.

[21] Torsten Grust, Jan Rittinger, and Jens Teubner. Why off-the-shelf rdbmss are better at xpath than you might expect. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 949–958, New York, NY, USA, 2007. ACM.

[22] Torsten Grust, Maurice Keulen van, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In J.-C. Freytag, P.C. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, editors, *Proceedings 2003 International Conference on Very Large Data Bases*, pages 524–535. Morgan Kaufmann Publishers, September 2003.

[23] Torsten Grust and Maurice van Keulen. Tree awareness for relational dbms kernels: Staircase join. In Henk Blanken, Torsten Grabs, Hans-Jrg Schek, Ralf Schenkel, and Gerhard Weikum, editors, *Intelligent Search on XML Data*, volume 2818 of *Lecture Notes in Computer Science*, pages 231–245. Springer Berlin / Heidelberg, 2003.

[24] Mena B. Habib and Maurice Keulen van. Improving named entity disambiguation by iteratively enhancing certainty of extraction, December 2011.

[25] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 1–9, New York, NY, USA, 2006. ACM.

[26] E. S. Hollander and M. van Keulen. Storing and querying probabilistic xml using a probabilistic relational dbms. In *4th International Workshop on Management of Uncertain Data, MUD 2010*, volume WP10-?? of *CTIT Workshop Proceedings Series*, Enschede, The Netherlands, September 2010. Centre for Telematics and Information Technology, University of Twente.

[27] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: a probabilistic database management system. In *SIGMOD Conference'09*, pages 1071–1074, 2009.

[28] E. Hung, L. Getoor, and V.S. Subrahmanian. Pxml: a probabilistic semistructured data model and algebra. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 467 – 478, march 2003.

[29] Edward Hung, Lise Getoor, and V. S. Subrahmanian. Probabilistic interval xml. *ACM Trans. Comput. Logic*, 8, August 2007.

[30] Tomasz Imieliński and Witold Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, September 1984.

[31] Florent Jousse, Rmi Gilleron, Isabelle Tellier, and Marc Tommasi. Conditional random fields for xml trees. In *In Proc. ECML Workshop on Mining and Learning in Graphs*, 2006.

[32] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query efficiency in probabilistic xml models. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 701–714, New York, NY, USA, 2008. ACM.

[33] Benny Kimelfeld, Yuri Kosharovsky, and Yehoshua Sagiv. Query evaluation over probabilistic xml. *The VLDB Journal*, 18:1117–1140, October 2009.

[34] Benny Kimelfeld and Yehoshua Sagiv. Matching twigs in probabilistic xml. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 27–38. VLDB Endowment, 2007.

[35] Christoph Koch, Lyublena Antova (cornell, Jiewen Huang (oxford, Goetz (cornell). Thomas Jansen, Ali Baran, and Sari Maybms. Maybms: A system for managing large uncertain and probabilistic databases. In *Managing and Mining Uncertain Data, chapter 6*. Springer-Verlag, 2008.

[36] Jasper Kuperus. Catching criminals by chance: a probabilistic approach to named entity recognition using targeted feedback, June 2012.

[37] Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. From region encoding to extended dewey: on efficient processing of xml twig pattern matching. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 193–204. VLDB Endowment, 2005.

[38] Clifford A. Lynch. When documents deceive: trust and provenance as new factors for information retrieval in a tangled web. *J. Am. Soc. Inf. Sci. Technol.*, 52(1):12–17, January 2001.

[39] Matteo Magnani and Danilo Montesi. A survey on uncertainty management in data integration. *J. Data and Information Quality*, 2(1):5:1–5:33, July 2010.

[40] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, USA, 1999.

[41] Andrew Nierman and H. V. Jagadish. Protdb: Probabilistic data in xml. In *In Proceedings of the 28th VLDB Conference*, pages 646–657. Springer, 2002.

[42] F. Panse, M. van Keulen, and N. Ritter. Indeterministic handling of uncertain decisions in duplicate detection. Technical Report TR-CTIT-10-21, Centre for Telematics and Information Technology University of Twente, Enschede, June 2010.

[43] Mingxi Wu Ravi Jampani, Fei Xu and Peter J. Haas Luis Perez, Chris Jermaine. The monte carlo database system: Stochastic analysis close to the data. In *ACM Transactions on Database Systems, Vol. 36, No. 3*, PODS '06. ACM, 2011.

[44] Pierre Senellart. Probabilistic XML: A data model for the Web, June 2012. Habilitation to supervise research.

[45] Pierre Senellart and Serge Abiteboul. On the complexity of managing probabilistic xml data. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 283–292, New York, NY, USA, 2007. ACM.

[46] Pierre Senellart and Asma Souihli. Proapprox: a lightweight approximation query processor over probabilistic trees. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1295–1298, New York, NY, USA, 2011. ACM.

[47] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[48] Paul Stapersma. A probabilistic xml database on top of maybms. In *Proceedings of the 13th Twente Student Conference on IT*. University of Twente, 2010.

[49] M. van Keulen. Managing uncertainty: The road towards better data interoperability. *IT - Information Technology*, 54(3):138–146, May 2012.

[50] Maurice van Keulen and Ander de Keijzer. Qualitative effects of knowledge rules and user feedback in probabilistic data integration. *The VLDB Journal*, 18:1191–1217, 2009.

[51] Maurice van Keulen, Ander de Keijzer, and Wouter Alink. A probabilistic xml approach to data integration. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *ICDE*, pages 459–470. IEEE Computer Society, 2005.

[52] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, Stanford InfoLab, August 2004.

Appendix A

Proofs

A.1 Closest node

Concept of closest node Given a context node c and a set of nodes ns in a tree-structure, we define a node $n \in ns$ to be *closest* –denoted as $n = closest(c, nd)$– if n is reached in the smallest amount of parent-steps. We introduce this concept in order to skip irrelevant nodes in a tree-structure. For example, if we request for the possibility parent of a node, we request for its the closest possibility node. Analogously, if we request for the parent axis of some node in a p-document, we request for the closest ordinary node of that node.

Evaluating the 'closest'-property with XA The XA approach –described in Section 2.3.2– encodes a tree-structure as $\langle pre, size \rangle$ tuples. Each such tuple represents one node in the tree-structure.

Let c be a context node and ns be a set of nodes. The following relation holds

$$n = closest(c, nd) \Leftrightarrow$$
$$pre(n) = \max\{n' : nd \mid pre(n') < pre(c) \leq pre(n') + size(n') \bullet pre(n')\}$$

Proof: By definition, n is part of the ancestor xis of c. It follows that the ancestor range condition applies to c and n:

$$n = closest(c, nd) \Rightarrow pre(n) < pre(c) \leq pre(n) + size(n)$$

We claim that the ancestor of c with the largest pre-order is closest to c. If not, a node n would be closest to c and there would exist a node n' with a larger or equal to the pre-order of n that is also ancestor of c such that $pre(n') \geq pre(n)$. Since n and n' are both ancestors of c, they lie on $\uparrow_n$ –the path from c to the root of the document. If we traverse $\uparrow_n$ from c to the root, the first node we discover other than c is n, since n is closest to c. It has to hold that all undiscovered nodes that lie on $\uparrow_n$ are ancestors of c and n. This includes node n'. It follows that n' resides in the ancestor axis of n. Hence, the ancestor range condition applies to n and n':

$$pre(n') < pre(n) \leq pre(n') + size(n')$$

A contradiction follows, since $pre(n') < pre(n)$ and $pre(n') \geq pre(n)$. We conclude that the closest node of c is the ancestor with the largest pre-order. □

Appendix B

SQL queries to perform glue process

B.1 Glue by Possibility Parent Reference

```
CREATE TABLE "next_ph" AS
(
  SELECT ph.* ,  sk.pre AS "sk"
  FROM "ph" ph
  ,  "sk" sk
  WHERE
      ph.posspre = sk.pre
)
```

Figure B.1: SQL query to evaluate one BB.PPR phase

B.2 Glue by Sandwich

A description of the SQL queries in this section is found in Section 11.3.

```
CREATE TABLE "next_ph" AS
(
  SELECT ph.* ,  sk.pre AS "sk"
  FROM "ph" ph
  ,  "sk" sk
  WHERE
      -- selects all ancestors of ph in sk
      ph.pre >= sk.pre
  AND ph.pre <= sk.pre+sk.size
  AND sk.pre =
                (SELECT  COALESCE
                  (
                    (
                      SELECT sk.pre
                      WHERE
                      NOT EXISTS
                      ( SELECT 1 WHERE
                        ( SELECT max(sk_copy.pre)
                          FROM "sk_copy" sk_copy
                          WHERE
                          sk_copy.depth = sk.depth+1
                          AND sk_copy.pre > sk.pre
                          AND ph.pre > sk_copy.pre
                          AND ph.pre <= sk_copy.pre+sk_copy.size
                          OR ph.pre = sk.pre
                        ) > -1
                      )
                      -- selects the don't care choice
                    ),( SELECT 0 WHERE ph.pre = 0)
                  )
                )
)
```

Figure B.2: SQL query to evaluate one BB.SW phase

B.3 Glue by Closest Descendant

```
CREATE TABLE "next_ph" AS
(
  SELECT ph.* ,  sk.pre AS "sk"
  FROM "ph" ph
  ,  "sk" sk
  WHERE
      -- selects all ancestors of ph in sk
      ph.pre >  sk.pre
  AND ph.pre <= sk.pre+sk.size
  AND sk.pre =
               (
                 SELECT COALESCE
                 (
                   (
                     SELECT max(sk_copy.pre)
                     FROM  "sk_copy" sk_copy
                     WHERE ph.pre >  sk_copy.pre
                     AND   ph.pre <= sk_copy.pre+sk_copy.size
                   ),0
                 )
               )
)
```

Figure B.3: SQL query to evaluate one BB.CD phase

B.4 Glue by Depth

```
CREATE TABLE "next_ph" AS
(
  SELECT ph.* ,  sk.pre AS "sk"
  FROM "ph" ph
  ,  "sk" sk
  WHERE
      sk.depth = x
  --selects all descendants of sk
  AND ph.pre > sk.pre
  AND ph.pre < sk.pre + sk.size
  AND ph.depth >= x
) UNION ALL (
  SELECT ph.* ,  sk.pre AS "sk"
  FROM "ph" ph
  ,  "sk" sk
  WHERE
      ph.pre = 0 -- refers to the don't care choice
  AND sk.depth < x
)
```

Figure B.4: SQL query to evaluate one BB.D phase

Appendix C

Benchmark Details

C.1 Query Workload

In this section, we present the query workload. The semantics of each query is provided.

Query category 1 (complex predicate) XPath queries in the first category all have a complex predicate that requests for a comparison of descendants with other descendants or other elements. We specify the semantics of this category as follows:

Query 1.1: *requests the title of all movies m for which an actor of m is also director of m.*

```
return //movie[(.//actor/name/string() = .//director/string())]/title
```

Query 1.2: *requests the title of all movies m for which an actor of m is also director.*

```
return //movie[(.//actor/name/string() = //director/string())]/title
```

Query 1.3: *requests the title of all movies m for which an director of m is also actor.*

```
return //movie[(.//director/string() = //actor/name/string())]/title
```

Query 1.4: *requests the title of all movies m for which an actor of m is also director and a director of m is also actor.*

```
    let $p := //movie[(.//actor/name/string() = //director/string())]/title
    let $q := //movie[(.//director/string() = //actor/name/string())]/title
return $p intersect $q
```

Query 1.5: *requests the title of all movies m for which the genre of m is 'Comedy' and there does not exists a possible world in which m has the genre 'Comedy' and 'Family'.*

```
    let $p := //movie[./genres/genre/string() = "Comedy"]/title
    let $q := //movie[./genres/genre/string() = "Family"]/title
return $p except possible ($p intersect $q)
```

Query 1.6: *requests the title of all movies m for which an director or actor of m are also director or actor in another movie m' that has the title 'Aventuras de las hermanas X'.*

```
    let $m_title := 'Aventuras de las hermanas X'
    let $m := //movie[./title/string() = $m_title]
    let $p := $m//actor/name; let $q := $m//director; let $pq := $p | $q
return //movie[(.//actor/name | .//director)/string() = $pq/string()]
    [compare(./title/string(), $m_title) != 0]/title
```

Query category 2 (foll/prec sibling) Each following XPath query in the second category increases the number of location steps that involve the following-siblining/preceding-sibling axis. We specify the semantics of this category as follows:

```
    let $genre := 'Horror'
```

Query 2.1: *requests for all occurrences of the genre 'Horror'.*

```
return //movie/descendant::genre[string() = $genre]
```

Query 2.2: *requests for all occurrences of a genre that is a following-sibling of the genre 'Horror'.*

```
return //movie/descendant::genre[string() = $genre]/following-sibling::genre
```

Query 2.3: *requests for all occurrences of a genre g that is a following-sibling of the horror genre g' for which holds that the preceding-sibling of g is also of the horror genre g''. Note that queries 2.2 and 2.3 have the same result.*

```
return //movie/descendant::genre[string() = $genre]/following-sibling::genre
    [./preceding-sibling::genre[string() = $genre]]
```

Query 2.4: *requests for all occurrences of a genre g that is a following-sibling of the horror genre g' for which holds that the preceding-sibling of g is also of the horror genre g'' and the following-sibling of genre g'' is the same genre as g.*

```
return //movie/descendant::genre[string() = $genre]/following-sibling::genre
```

```
[./preceding-sibling::genre[string() = $genre]/
following-sibling::genre/string() = ./string()]
```

Query category 3 (anc/desc) Each following XPath query in the third category increases the number of location steps that involve the ancestor/descendant axis. We specify the semantics of this category as follows:

```
let $genre := 'Comedy'
```

Query 3.1: *requests for all movies that have the genre 'Comedy'.*

```
return //genre[./string() = $genre]/ancestor::movie
```

Query 3.2: *requests for all titles of movies that have the genre 'Comedy'.*

```
return //genre[./string() = $genre]/ancestor::movie/descendant::title
```

Query 3.3: *requests for all movies that have a title of a movie with the genre 'Comedy'. Note that queries 3.1 and 3.3 have the same result if all movies in the result of 3.2 have a title.*

```
return //genre[./string() = $genre]/ancestor::movie/descendant::title
   /ancestor::movie
```

Query 3.4: *requests for all titles of movies that have a title of a movie with the genre 'Comedy'. Note that queries 3.2 and 3.4 have the same result.*

```
return //genre[./string() = $genre]/ancestor::movie/descendant::title
   /ancestor::movie/descendant::title
```

Query category 4 (par/child) Each following XPath query in the fourth category increases the number of location steps that involve the parent/child axis. We specify the semantics of this category as follows:

```
let $actor_name := 'Solomon, Regina'
```

Query 4.1: *requests for the role of all actors a of which the name of a is 'Solomon, Regina'.*

```
return //actor[name/string()=$actor_name]/child::role
```

Query 4.2: *requests for the parent of the role of all actors a of which the name of a is 'Solomon, Regina'. Since the parent of a role is the actor, query 4.2 requests for actors a of which the name of a is 'Solomon, Regina'*

```
return //actor[name/string()=$actor_name]/child::role/parent::*
```

Query 4.3: *requests for the child of the parent of the role of all actors a of which the name of a is 'Solomon, Regina'. Query 4.3 requests for the same result as query 4.1.*

```
return //actor[name/string()=$actor_name]/child::role/parent::*/child::role
```

Query 4.4: *requests for the parent of the child of the parent of the role of all actors a of which the name of a is 'Solomon, Regina'. Query 4.4 requests for the same result as query 4.2.*

```
return //actor[name/string()=$actor_name]/child::role/parent::*/child::role/parent::*
```

Query category 5 (simple predicate) XPath queries in the fifth category all have a simple predicate that requests for the existence of some element in one axis. We specify the semantics of this category as follows:

Query 5.1: *requests the title of movies m in which a director 'Thomas, Ralph (I)' directs and the genre of m is 'Comedy'.*

```
//genre[./string() = 'Comedy']/ancestor::movie[directors/director
   /string() = 'Thomas, Ralph (I)']/descendant::title
```

Query 5.2: *requests all ancestors of an actor a of which the name of a is 'Mehaffey, Blanche'.*

```
//actor[name/string()='Mehaffey, Blanche']/ancestor::*
```

Query 5.3: *requests all horror genres g that are following-sibling of a comedy genre g'.*

```
//movie/genres/genre[string() = 'Comedy']
   /following-sibling::genre[string() = 'Horror']
```

Query 5.4: *requests all descendants of movie m for which the title is 'Trials of a Movie Cartoonist, The'.*

```
//movie[title/string() = 'Trials of a Movie Cartoonist, The']/descendant::*
```

Query 5.5: *requests the title of all movies in which a director d has a name that starts with an 'A'.*

```
//director[substring(./string(),1,1) = 'A']/ancestor::movie/title
```

C.2 Indices for query evaluation

Figures C.1, C.2, C.3 and C.4 present the indices and set commands that we used to conduct our experiments. For simplicity, we refer to the set commands and indices used to perform a query as the database setting for that query. Rows hold database settings per query and columns hold database settings per query evaluation approach. The first column holds the database settings to evaluate t-queries. The second column holds the database settings to evaluate tg-queries on the flesh with a materialized view for the extended inlining rules. The third column holds the database settings to evaluate tg-queries on the flesh with a materialized view for the traditional inlining rules. Other columns hold the database settings to evaluate tg-queries that use the glue process specified in the column name.

We specify a database setting as a series of indices followed by a set of set operations. Indices are denoted as $I(cs)$ ord $CI(cs)$. The former denotes an unclustered B-tree index on a set of columns cs; the latter denotes a clustered B-tree index for a set of columns. Set commands are one of the following: $mergj, hashj, nestl, seqs$. Each occurrence of a set command denotes the disabling of that command.

Columns are abbreviated as follows:

1. 'a_x': refers to the column that holds property x for element kind 'actor'.
2. 'as_x': refers to the column that holds x for element kind 'actors'.
3. 'd_x': refers to the column that holds x for element kind 'director'.
4. 'ds_x': refers to the column that holds x for element kind 'directors'.
5. 'g_x': refers to the column that holds x for element kind 'genre'.
6. 'gs_x': refers to the column that holds x for element kind 'genres'.
7. 'l_x': refers to the column that holds x for element kind 'location'.
8. 'ls_x': refers to the column that holds x for element kind 'locations'.
9. 'm_x': refers to the column that holds x for element kind 'movie'.
10. 'ms_x': refers to the column that holds x for element kind 'movies'.
11. 'n_x': refers to the column that holds x for element kind 'name'.
12. 'o_x': refers to the column that holds x for element kind 'otherinfo'.
13. 'p_x': refers to the column that holds x for element kind 'plot'.
14. 'ps_x': refers to the column that holds x for element kind 'plots'.
15. 'r_x': refers to the column that holds x for element kind 'role'.
16. 't_x': refers to the column that holds x for element kind 'title'.
17. 'y_x': refers to the column that holds x for element kind 'year'.

A property is one of the following:

1. 'id': refers to the pre-order of an element. Pre-order is discussed in Section 2.3.2.
2. 'par': refers to the pre-order of the parent node of an element.
3. 'size': refers to the number of descendants of an element.
4. 'pcdata': holds the PCData for a text node.
5. 'posspre': refers to the pre-order of the possibility parent. The concept of a possibility parent is discussed in Section 10.2.1.

Q	t-queries	FLESH	FLESH_TI	PC.PPR	PC.CD_TI	PC.CD & BB.CD	BB.PPR
101	I(a_id), I(d_id), I(m_id, m_size), I(n_pcdata), I(t_par, t_pcdata) [hashj., mergj., seqs.]	I(a_id), I(d_id, d_pcdata) I(n_pcdata), I(m_id, m_size), I(t_pcdata), [mergj., hashj., seqs.]	I(m_size, m_id), I(m_t_id, m_t_pcdata), I(a_id), CI(d_id), I(a_n_pcdata) [hashj., mergej., seqs.]	I(a_id), I(d_id), I(m_id), I(n_pcdata), I(t_pcdata) [hashj., mergj., seqs.]	I(m_size, m_id), I(m_t_id, m_t_pcdata), I(a_id), CI(d_id), I(a_n_pcdata) [hashj., mergej., seqs.]	I(m_id, m_size), I(d_id), I(a_id), I(n_pcdata, n_id, n_par), I(t_pcdata)	I(a_id), I(d_id), I(m_id), I(n_pcdata), I(t_pcdata) [hashj., mergj., seqs.]
102	I(a_id), I(d_pcdata), I(m_par, t_pcdata) I(m_size), I(n_pcdata, n_par), [mergj., hashj., seqs.]	I(n_pcdata), I(m_id, m_size), I(t_par, t_pcdata), I(a_id), I(d_pcdata) [mergj., hashj., seqs.]	I(m_size, m_t_pcdata, m_t_id), I(a_id, a_n_pcdata), I(d_pcdata) [mergej.]	I(n_pcdata), I(m_size), CI(t_pcdata), I(a_id), I(d_pcdata) [hashj., mergj., seqs.]	I(m_size, m_t_pcdata, m_t_id), I(a_id, a_n_pcdata), I(d_pcdata) [mergej.]	I(n_pcdata), I(m_size), I(t_id, t_pcdata), I(a_id), I(d_id) [mergj., hashj., seqs.]	I(a_id), I(t_par, t_posspre), I(m_id, m_size), I(n_pcdata, n_posspre, n_par), I(d_pcdata, d_posspre) [hashj., mergj., seqs.]
103	I(a_id), I(d_id), I(n_pcdata, n_par), I(n_size), I(t_pcdata) [seqs., mergj.]	I(n_pcdata), I(m_size), I(t_pcdata), I(a_id), I(d_id, d_pcdata) [mergj., hashj., seqs.]	I(m_t_id, m_id, m_size), I(a_n_pcdata), I(d_id, d_pcdata) [mergej., hashj., seqs.]	I(n_pcdata, n_par, n_posspre), I(m_id, m_size), I(t_par), I(a_id), I(d_pcdata, d_id, d_posspre) [hashj., mergj., seqs.]	I(m_id, m_t_id, m_t_pcdata, m_size), I(a_id, a_n_pcdata), I(d_pcdata) [mergej.]	I(n_pcdata, n_id, n_par), I(m_id, m_size), I(t_par, t_id), I(a_id), I(d_pcdata, d_id) [hashj., mergj., seqs.]	I(n_pcdata), I(m_size), I(t_par, t_posspre, t_pcdata), I(a_id), I(d_id, d_posspre) [hashj., mergj., seqs.]
104	I(a_id), CI(d_id, d_pcdata), I(d_pcdata, d_id), I(m_id), CI(m_id), I(n_par), I(n_pcdata), I(t_id), I(t_id, t_par), I(t_par, t_id) [hashj., mergj., seqs.]	I(d_id), I(n_pcdata), I(d_id), I(t_id, t_par, t_pcdata), I(n_id, n_par, n_pcdata), I(m_id, m_par), I(t_pcdata, t_id, t_par), I(a_id), I(m_size, m_id) [hashj., mergj., seqs.]	I(d_id), I(d_id), I(m_t_id), I(a_n_pcdata), I(a_id, a_n_id), I(m_t_id, m_id) [hashj., mergej., seqs.]	I(m_id), I(d_pcdata, d_id), I(d_id, d_pcdata), I(a_id), I(n_par), I(n_pcdata), I(t_id), I(t_id, t_par), CI(t_par) [hashj., mergj., seqs.]	I(d_id), I(d_id), I(m_t_id), I(a_n_pcdata), I(a_id, a_n_id), I(m_t_id, m_id) [hashj., mergej., seqs.]	I(n_pcdata), I(d_id), I(t_id, t_par, t_pcdata), I(n_id, n_par, n_pcdata), I(m_id, m_par), I(t_pcdata, t_id, t_par), I(a_id), I(m_size, m_id) [hashj., mergj., seqs.]	I(m_id), I(d_pcdata, d_id), I(d_id, d_pcdata), I(a_id), I(n_par), I(n_pcdata), I(t_id), I(t_id, t_par), CI(t_par) [hashj., mergj., seqs.]
105	I(g.par) I(gs.par), I(m_id), [hashj., seqs.]	I(m_id), I(g_pcdata, g_par, g_id), I(gs_par, gs_id) [seqs., mergj.]	I(m_gs_id), I(g_par, g_id, g_pcdata) [hashj., mergej., seqs.]	I(m_id), I(g_pcdata, g_par), I(gs_par, gs_id), [hashj.]	I(m_gs_id), I(g_par, g_id, g_pcdata) [hashj., mergej., seqs.]	I(m_id), I(g_pcdata, g_par, g_id), I(gs_par, gs_id) [seqs., mergj.]	I(m_id), I(g_pcdata, g_par), I(gs_par, gs_id), [hashj.]
106	I(a_id), I(a_id, a_par), I(d_pcdata, d_id), I(m_id), I(n_id), CI(n_par), I(t_pcdata, t_par), []	I(a_id), CI(n_par), I(t_pcdata, t_par), I(m_id), I(a_id, a_par), I(n_id), CI(t_pcdata, t_par), I(m_id), I(d_pcdata, d_id), [mergj.]	I(m_t_id, m_t_pcdata, m_size, m_id) [hashj., mergej., seqs.]	I(m_size), I(t_par, t_posspre, t_pcdata) [nestl., mergj.]	I(m_t_id, m_t_pcdata, m_size, m_id) [hashj., mergej., seqs.]	I(a_id), CI(n_par), I(t_pcdata, t_par), I(m_id), I(a_id, a_par), I(n_id), CI(t_pcdata, t_par), I(d_pcdata, d_id), [mergj.]	I(m_size), I(t_par, t_posspre, t_pcdata) [nestl., mergj.]

Figure C.1: Indices used to evaluate XPath queries of query workload category 1

Q	t-queries	FLESH	FLESH_TI	PC.PPR	PC.CD_TI	PC.CD & BB.CD	BB.PPR
201	I(g_id), I(m_size) [seqs.]	I(g_pcdata, g_id), I(m_id) [mergj., hashj.]	I(m_size), I(g_pcdata, g_id) [mergej.]	I(m_size, m_id), I(g_pcdata, g_id, g_posspre) [hashj., mergj., nestl.]	I(m_size), I(g_pcdata, g_id) [mergej.]	I(g_pcdata, g_id), I(m_id) [mergj., hashj.]	I(m_id), I(g_pcdata, g_id, g_posspre) [hashj., mergj., seqs.]
202	I(m_id, m_size), I(g_pcdata, g_size), I(g_par) [mergj.]	I(m_id, m_size), I(g_id), I(g_par, g_id, g_size, g_pcdata) [mergj., hashj., seqs.]	I(m_id), I(g_par, g_id), I(g_id, g_pcdata, g_par) [mergej., hashj., seqs.]	I(m_size, m_id), I(g_par), I(g_id, g_size, g_posspre, g_pcdata, g_par), I(m_size, m_id), I(g_par, g_posspre, g_id), [mergj., hashj., seqs.]	CI(m_id, m_size), I(g_pcdata), CI(g_par) [mergej., hashj., seqs.]	I(m_id, m_size), I(g_pcdata, g_par, g_size), I(g_par) [mergj., hashj., seqs.]	I(m_id), CI(g_par), I(g_pcdata) [hashj., mergj., seqs.]
203	I(m_id), CI(g_par), I(g_pcdata, g_id) [mergj.]	I(m_id), CI(g_par), I(g_pcdata, g_id) [hashj.]	I(m_id, m_size), I(g_pcdata), I(g_par) [mergej., hashj., seqs.]	I(m_id), CI(g_pcdata, g_size), I(g_par, g_id, g_posspre), I(g_id, g_par, g_posspre, g_size, g_pcdata) [mergj., hashj., seqs.]	I(m_id, m_size), I(g_pcdata), I(g_par) [mergej., hashj., seqs.]	I(m_id), I(g_id, g_pcdata, g_par, g_size), I(g_par, g_id), I(g_par, g_id, g_pcdata, g_size) [hashj., mergj., seqs.]	I(m_id), I(g_par), I(g_pcdata, g_id) [hashj.]
204	I(m_id), CI(g_par), I(g_pcdata, g_id) [hashj.]	I(m_id), I(g_id, g_pcdata, g_par, g_size), I(g_par, g_pcdata, g_id), I(g_size, g_par, g_pcdata, g_id) [mergj., hashj., seqs.]	I(m_id, m_size), CI(g_pcdata), CI(g_par) [mergej., hashj., seqs.]	CI(g_par), I(m_size), I(g_id, g_pcdata, g_par, g_posspre, g_size), I(g_par, g_id, g_posspre, g_pcdata), I(g_id, g_par, g_posspre, g_size, g_pcdata) [mergj., hashj., seqs.]	I(m_id, m_size), CI(g_pcdata), CI(g_par) [mergej., hashj., seqs.]	I(g_id, g_pcdata), I(m_id, m_size), I(g_par), I(g_par, g_id, g_pcdata), I(g_size, g_par, g_pcdata, g_id) [mergj., hashj., nestl.]	CI(g_par), I(m_size), I(g_id, g_pcdata, g_par, g_posspre), I(g_par, g_id, g_posspre, g_pcdata), I(g_id, g_par, g_posspre, g_size, g_pcdata) [mergj., hashj., seqs.]
301	I(g_id, g_pcdata), I(m_id) [nestl., hashj.]	I(m_id, m_size), I(g_id, g_pcdata) [hashj., nestl.]	I(g_id, g_pcdata), I(m_id, m_size) [nestl., hashj., mergej.]	I(m_size, m_id), I(g_pcdata) [seqs., hashj.]	I(g_id), I(m_id, m_size) [mergej., hashj., nestl.]	I(m_id, m_size), I(g_id, g_pcdata) [hashj., nestl.]	CI(g_pcdata, g_posspre), I(m_id, m_size) []
302	I(t_id), I(g_id, g_pcdata) [hashj., mergj., seqs.]	I(t_id), I(m_id, m_size), CI(g_pcdata, g_id) [mergj., seqs.]	I(m_id, m_size, m_t_id), I(g_id) [hashj., mergej., seqs.]	I(t_id), I(m_id, m_size), I(g_posspre) [seqs., mergj.]	I(m_id, m_size, m_t_id), I(g_id) [hashj., mergej., seqs.]	I(t_id), I(m_id, m_size), CI(g_pcdata, g_id) [mergj., seqs.]	I(t_id), I(m_id, m_size), I(g_posspre) [mergj.]
303	I(g_id, g_pcdata), I(m_id), I(t_id), I(m_size) [hashj., nestl.]	I(t_id), I(m_size), I(m_id), I(g_id, g_pcdata) [nestl.]	I(g_id), CI(m_id, m_t_id) [seqs., hashj., mergej.]	I(t_id, t_posspre), I(m_id), I(g_id) [seqs., mergj., nestl.]	I(g_id), CI(m_id, m_t_id) [seqs., hashj., mergej.]	I(t_id), I(m_id, m_size), I(g_id, g_pcdata) [seqs., nestl.]	I(t_id, t_posspre), I(m_id), I(g_id) [seqs., mergj., nestl.]
304	I(g_pcdata, g_id), I(m_id, m_size), I(m_size), I(t_id) [nestl.]	I(g_pcdata, g_id), I(m_id, m_size), I(t_id), I(m_size), I(t_id) [nestl.]	CI(g_id), CI(m_id, m_t_id, m_size) [seqs., hashj., mergej.]	I(g_pcdata, g_id), I(m_id, m_size), CI(m_id), I(t_id), I(t_id, t_posspre) [nestl., seqs., hashj.]	CI(g_id), CI(m_id, m_t_id, m_size) [seqs., hashj., mergej.]	I(g_pcdata, g_id), I(m_id, m_size), I(t_id) [nestl.]	I(t_id), I(m_id, m_size), I(t_id), CI(g_pcdata, g_posspre) [seqs., hashj., mergj.]

Figure C.2: Indices used to evaluate XPath queries of query workload categories 2 & 3

Q	t-queries	FLESH	FLESH_TI	PC.PPR	PC.CD_TI	PC.CD & BB.CD	BB.PPR
401	I(n_pcdata), I(a_id), I(r.par) [hashj., mergj., seqs.]	I(n_pcdata, n_par), I(a_id), CI(r_par) [hashj., mergj., seqs.]	CI(a_n_pcdata, a_r_id) [hashj., mergej., seqs.]	I(n_pcdata), I(r_par), CI(a_id) [hashj., mergj., seqs.]	CI(a_n_pcdata, a_r_id) [hashj., mergej., seqs.]	I(n_pcdata, n_par), I(a_id), CI(r_par) [hashj., mergj., seqs.]	I(n_pcdata, n_par), I(a_id), CI(r_par) [hashj., mergj., seqs.]
402	I(n_pcdata), I(a_id), CI(r_par), [hashj., mergj., seqs.]	I(n_pcdata), I(a_id), I(r_par) [mergj., hashj., seqs.]	CI(a_n_pcdata) [nestl., mergej., hashj.]	I(n_pcdata, n_par), I(a_id), I(r_par), [seqs., mergj., hashj.]	CI(a_n_pcdata) [nestl., mergej., hashj.]	I(n_pcdata), I(a_id), CI(r_par) [hashj., mergj., seqs.]	I(n_pcdata, n_par), I(a_id), I(r_par), CI(a_posspre) [hashj., mergj., seqs.]
403	I(a_id), I(n_pcdata), CI(r_par), I(a_id), I(r_id) [hashj., mergj., seqs.]	I(n_pcdata), I(a_id), I(r_id), I(r_par) [mergj., hashj., seqs.]	CI(a_n_pcdata) [seqs., hashj., mergej.]	I(n_pcdata, n_par), I(r_par) [seqs., hashj., mergj.]	CI(a_n_pcdata) [seqs., hashj., mergej.]	I(a_id), I(n_pcdata), CI(r_par), I(r_id) [hashj., mergj., seqs.]	I(n_pcdata), I(a_id), CI(r_par), I(r_par), [hashj., mergj., seqs.]
404	I(a_id), I(n_pcdata), CI(r_par), CI(r_par), [hashj., seqs., mergj.]	I(n_pcdata), I(a_id), I(r_par), [mergj., hashj., seqs.]	CI(a_n_pcdata) [seqs., hashj., mergej.]	CI(n_pcdata, n_par), I(r_par, r_id) [seqs., hashj., mergj.]	CI(a_n_pcdata) [seqs., hashj., mergej.]	I(n_pcdata), CI(r_par), I(a_id), CI(r_par), [hashj., seqs., mergj.]	CI(n_pcdata, n_par), I(a_id), I(r_id), I(r_par), [hashj., mergj., seqs.]

Figure C.3: Indices used to evaluate XPath queries of query workload category 4

Q	t-queries	FLESH	FLESH_TI	PC.PPR	PC.CD_TI	PC.CD & BB.CD	BB.PPR
501	I(g_id), CI(m_id, m_size), CI(ds_id), CI(d_pcdata), I(t_id) [seqs., hashj., mergj.]	I(ds_id), I(m_id), I(t_id, t_pcdata), I(g_pcdata, g_id), I(d_pcdata, d_par) [nestl., hashj., mergj.]	I(g_id), I(m_ds_id, m_id, m_t_pcdata, m_t_id, m_size), I(d_pcdata, d_par) [seqs., hashj., mergej.]	I(m_size), I(d_pcdata) [nestl., hashj., mergj.]	CI(g_id, g_pcdata), CI(d_pcdata), I(m_t_id, m_ds_id, m_t_pcdata) [nestl., hashj., mergej.]	I(g_id), CI(m_id, m_size), CI(ds_id), CI(d_pcdata), I(t_id) [seqs., hashj., mergj.]	I(g_pcdata, g_id), I(m_id), I(d_pcdata, d_id), I(t_id) [seqs., hashj.]
502	I(a_id), I(m_id), I(n_pcdata, n_par) [seqs.]	I(a_id), I(m_id), I(n_pcdata, n_par) [seqs.]	I(a_name_pcdata, a_name_id) [seqs., mergej., hashj.]	I(a_id), I(m_id), I(n_pcdata, n_par) [seqs.]	I(a_name_pcdata, a_name_id) [seqs., mergej., hashj.]	I(a_id), I(m_id), I(n_pcdata, n_par) [seqs.]	I(a_id), I(m_id), I(n_pcdata, n_par) [seqs.]
503	I(m_id), I(gs_id), I(g_pcdata) [mergj., seqs.]	I(m_id), I(g_pcdata), I(g_par, g_size, g_id, g_pcdata), I(gs_id, gs_par) [mergj., hashj., seqs.]	I(g_pcdata), I(g_par, g_id, g_pcdata, g_size), I(m_gs_id) [mergej., hashj., seqs.]	I(m_id), I(gs_par, gs_id), I(g_size, g_par, g_pcdata, g_posspre, g_id), I(g_par, g_posspre, g_pcdata) [hashj., mergj., seqs.]	I(g_pcdata), I(g_par, g_id, g_pcdata, g_size), I(m_gs_id) [mergej., hashj., seqs.]	I(m_id), I(gs_id), I(g_pcdata), [mergj., seqs.]	I(m_id), I(gs_par, gs_id), I(g_size, g_par, g_pcdata, g_posspre, g_id), I(g_par, g_posspre, g_pcdata) [hashj., mergj., seqs.]
504	I(t_id), I(otherinfo_id), I(g_id), I(gs_id), I(n_id), I(r_id), I(a_id), I(as_id), I(d_id), I(ds_id), I(l_id), I(ls_id), I(k_id), I(ks_id), I(p_id), I(ps_id), I(y_id), CI(m_id), I(m_id, m_size), I(t_pcdata, t_par) [hashj., mergj., seqs.]	I(t_id), I(otherinfo_id), I(g_id), I(gs_id), I(n_id), I(r_id), I(a_id), I(as_id), I(d_id), I(ds_id), I(l_id), I(ls_id), I(k_id), I(ks_id), I(p_id), I(ps_id), I(y_id), CI(m_id), I(m_id, m_size), I(t_pcdata, t_par) [hashj., mergj., seqs.]	I(m_t_id), I(m_otherinfo_id), I(g_id), I(m_gs_id), I(a_n_id), I(a_r_id), I(a_id), I(m_as_id), I(d_id), I(m_ds_id), I(l_id), I(m_ls_id), I(k_id), I(m_ks_id), I(p_id), I(m_ps_id), I(m_y_id), I(m_id), CI(m_id, m_size), I(m_t_pcdata, m_par) [hashj., mergej.]	I(t_id), I(otherinfo_id), I(g_id), I(gs_id), I(n_id), I(r_id), I(a_id), I(as_id), I(d_id), I(ds_id), I(l_id), I(ls_id), I(k_id), I(ks_id), I(p_id), I(ps_id), I(y_id), CI(m_id), I(m_id, m_size), I(t_pcdata, t_par) [mergj., seqs.]	I(m_t_id), I(m_otherinfo_id), I(g_id), I(m_gs_id), I(a_n_id), I(a_r_id), I(a_id), I(m_as_id), I(d_id), I(m_ds_id), I(l_id), I(m_ls_id), I(k_id), I(m_ks_id), I(p_id), I(m_ps_id), I(m_y_id), I(m_id), CI(m_id, m_size), I(m_t_pcdata, m_par) [hashj., mergej.]	I(t_id), I(otherinfo_id), I(g_id), I(gs_id), I(n_id), I(r_id), I(a_id), I(as_id), I(d_id), I(ds_id), I(l_id), I(ls_id), I(k_id), I(ks_id), I(p_id), I(ps_id), I(y_id), CI(m_id), I(m_id, m_size), I(t_pcdata, t_par) [hashj., mergj., seqs.]	I(t_id), I(otherinfo_id), I(g_id), I(gs_id), I(n_id), I(r_id), I(a_id), I(as_id), I(d_id), I(ds_id), I(l_id), I(ls_id), I(k_id), I(ks_id), I(p_id), I(ps_id), I(y_id), CI(m_id), I(m_id, m_size), I(t_pcdata, t_par) [hashj., mergj., seqs.]
505	I(d_id), I(m_size), I(t_par, t_pcdata) [mergj., seqs., nestl.]	I(m_id, m_size), I(t_id, t_pcdata) [seqs., mergj., hashj.]	I(d_id), I(m_t_id, m_t_pcdata) [mergej., nestl., seqs.]	I(t_pcdata), I(m_id) [seqs., hashj.]	I(d_id), I(m_t_id, m_t_pcdata) [mergej., nestl., seqs.]	I(m_id, m_size), I(t_id, t_pcdata) [seqs., mergj., hashj.]	I(t_pcdata), I(m_id) [seqs., hashj.]

Figure C.4: Indices used to evaluate XPath queries of query workload category 5

C.3 DTD of IMDb data set

For our experimental validation in Chapter 12, we conduct a performance study on three different test sets that all derive from the IMDb data set. We describe these test sets in in Section 12.2.1. In Figure C.5, we present the DTD of the IMDb data set.

```
<!ELEMENT movies(movie*)>
<!ELEMENT movie (title,otherinfo,genres,
                 actors,directors,locations,keywords,plots,year)>
<!ELEMENT title (#PCDATA)>        <!ELEMENT otherinfo (#PCDATA)>
<!ELEMENT actors (actor*)>        <!ELEMENT actor (name?,role?)>
<!ELEMENT name (#PCDATA)>         <!ELEMENT role (#PCDATA)>
<!ELEMENT genres (genre*)>        <!ELEMENT genre (#PCDATA)>
<!ELEMENT directors (director*)>  <!ELEMENT director (#PCDATA)>
<!ELEMENT locations (location*)>  <!ELEMENT location (#PCDATA)>
<!ELEMENT keywords (keyword*)>    <!ELEMENT keyword (#PCDATA)>
<!ELEMENT plots (plot*)>          <!ELEMENT plot (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

Figure C.5: DTD schema of the IMDb data set

C.4 Normalized speedups for XPath query evaluation by a URDBMS

This section provides

C.4.1 Experimental results for query evaluation on increasing data set size

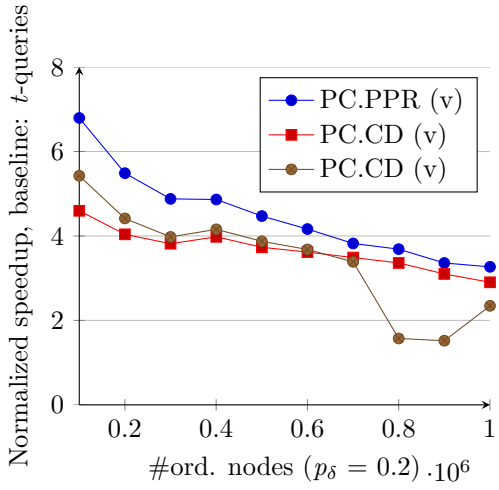
We show the performance results of Figures 12.6b and 12.6a in more detail in Figures C.6a, C.6b, C.6c, C.6d and C.6e. These figures show normalized speedup for *tg*-query evaluation compared to *t*-query evaluation per query category for an increasing data set size. We selected the *tg*-query categories that use glue process PC.PPR or PC.CD_*TI*. These glue processes have the best performance in one of the five categories as shown in Figure 12.6a. We added the PC.CD glue method such that we get insight in the effects of inlining. We observe the following:

1. In general, the speedup of most *tg*-query categories is constant. This indicates that the evaluation of *tg*-queries has the same complexity as the evaluation of *tg*-queries. This observation makes a comparison between the evaluation of *t*-queries compared to the evaluation of *tq*-queries more easy since complexity of both query evaluation approaches is the same in all cases where speedup is constant. Exceptions are shown in Figures C.6a, and C.6c which we discuss next.

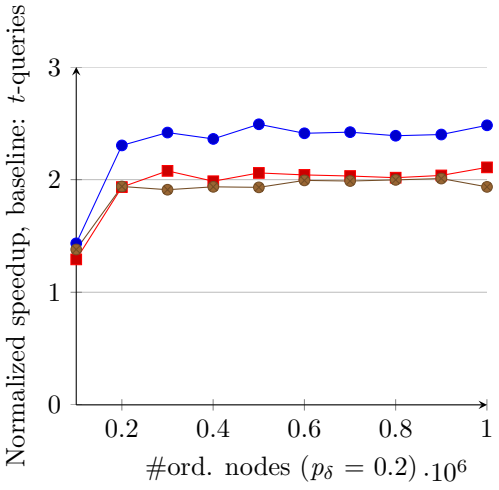
2. Figure C.6a: we observe that for increasing data set size, the speedup decreases. The speedup of all *tg*-queries seem to converge to a constant speedup of 3. This decrease in speedup can be caused by an inefficient query execution plan since we already identified query evaluation plans for the first category with unmaterialized views to be inefficient. Speedup for materialized views do show a constant speedup, however, these speedups are not illustrated.

3. Figures C.6c and C.6d show a clear difference in performance behaviour between PC.CD and PC.CD_*TI* in advantage of PC.CD_*TI*. Since these two methods use a different set of inlining rules, we attribute this performance difference to the loosening in the inlining rules.

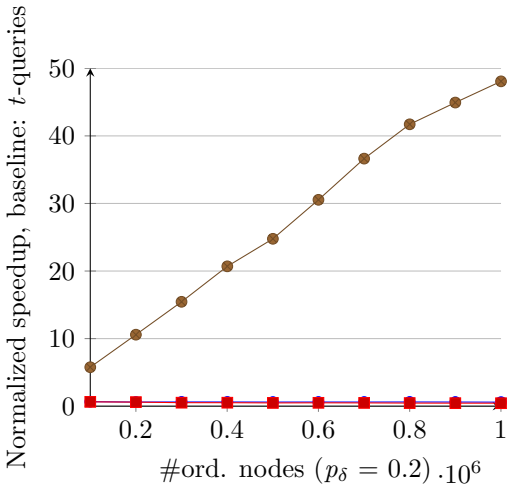
4. Figures C.6c: we observe a linear increase in speedup for *tg*-queries that use the glue method PC.CD_*TI*. This behaviour indicates a linear reduction in complexity caused by the loosening in the inlining rules.



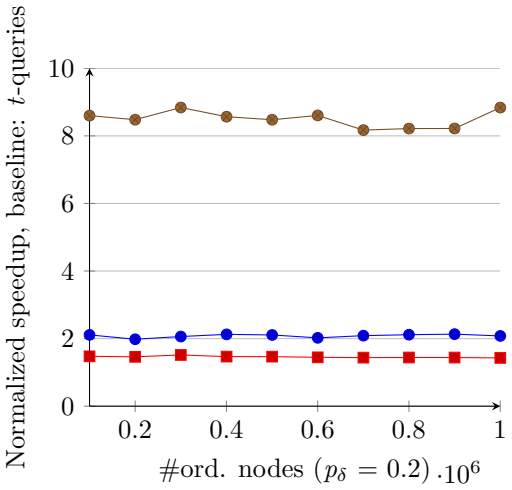
(a) Results for category 1 (Complex)



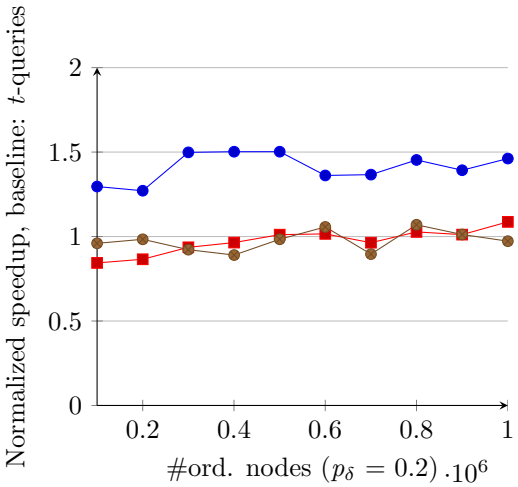
(b) Results for category 2 (Foll/Prec Sibling)



(c) Results for category 3 (Anc/Desc)



(d) Results for category 4 (Par/Child)



(e) Results for category 5 (Simple)

Figure C.6: Experimental results for increasing data set sizes

C.4.2 Experimental results for query evaluation on increasing amounts of uncertainty

We show the performance results of Figures 12.7b and 12.7a in more detail in Figures C.7a, C.7b, C.7c, C.7d and C.7e. These figures show normalized speedup for *tg*-query evaluation compared to *t*-query evaluation per query category for an increasing amount of uncertainty. We selected the *tg*-query categories that use glue process PC.PPR or PC.CD_*TI*. These glue processes have the best performance in one of the five categories as shown in Figure 12.6a. We added the PC.CD glue method such that we get insight in the effects of inlining. We observe the following:

1. In general, the speedup of most *tg*-queries is constant. This indicates that the evaluation of *tg*-queries has the same complexity as the evaluation of *t*-queries. This observation makes a comparison between the evaluation of *t*-queries compared to the evaluation of *tq*-queries more easy since complexity of both query evaluation approaches is the same in all cases where speedup is constant. Exceptions are shown in Figures C.7a, C.7b and C.7c which we discuss next.

2. Figure C.7a: we observe that speedup for all three *tg*-query categories increases for uncertainty amounts of 80% and more. Materialized views show the same behaviour.

3. Figure C.7b: we observe similar behaviour for the speedup of all three *tg*-query categories. However, this behaviour does not satisfy a pattern.

4. Figure C.7c: we observe two outliers in the speedup of *tg*-query category PC.CD_*TI*. A reduced form of these outliers are also present in the speedup behaviour of the other two *tg*-query categories. These outliers are likely to be caused by inefficient query execution plans since we already identified query evaluation plans for the first category with unmaterialized views to be inefficient. Speedup for materialized views do show a constant speedup, however, these speedups are not illustrated.

5. Figures C.7c and C.7d show a clear difference in performance behaviour between PC.CD and PC.CD_*TI* in advantage of PC.CD_*TI*. Since these two methods use a different set of inlining rules, we attribute this performance difference to the loosening in the inlining rules.

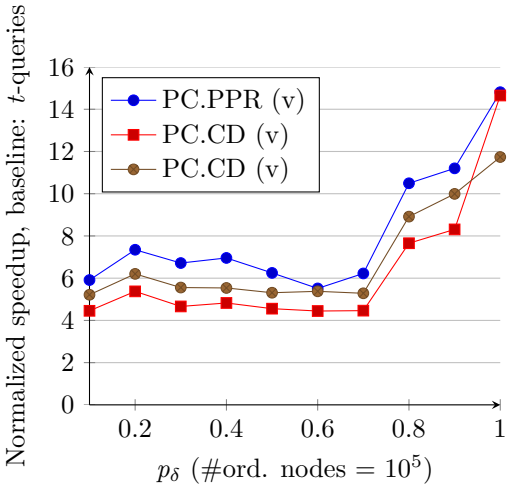
C.4.3 Experimental results for query evaluation on a real world uncertain test set

We show the performance results of Figures 12.8b and 12.8a in more detail in Figures C.8a, C.8b, C.8c, C.8d and C.8e. These figures show normalized speedup for *tg*-query evaluation compared to *t*-query evaluation per query category for an increasing movie title threshold. This threshold is some parameter that determines the amount of uncertainty in the document. We selected the *tg*-query categories that use glue process PC.PPR, PC.CD or PC.CD_*TI* since we also selected these glue processes for previous experiments. We observe the following:

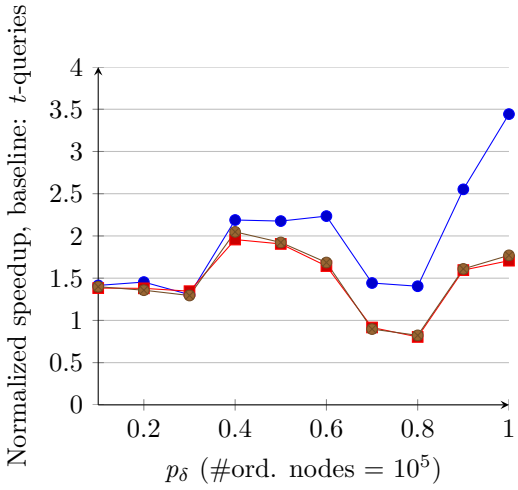
1. In general, the speedup of most *tg*-queries is constant. This indicates that the evaluation of *tg*-queries has the same complexity as the evaluation of *t*-queries.

2. For all categories, PC.CD_*TI* has a lower normalized speedup than PC.CD. This indicates that each query category profits from the loosening of the inline rules.

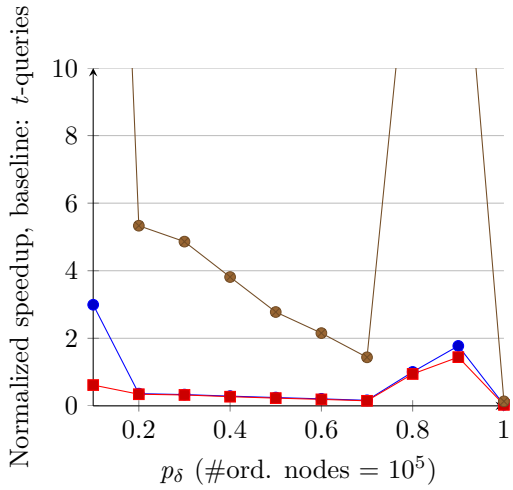
3. Figure C.8c: we observe that *tg*-queries that use PC.CD (v) has an outlier for a movie title threshold of 0.4. This outlier is the result of the query planner to selects a different query execution plan than for the other thresholds.



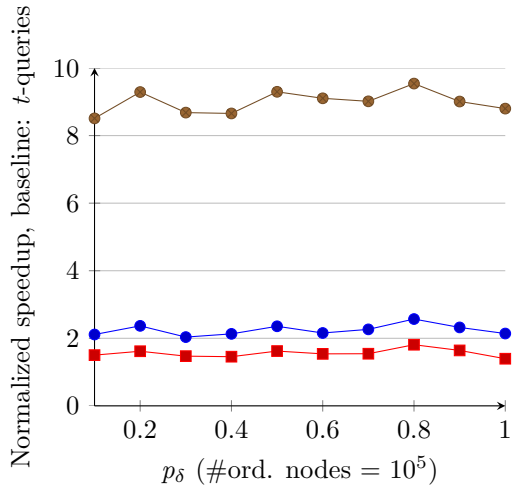
(a) Results for category 1 (Complex)



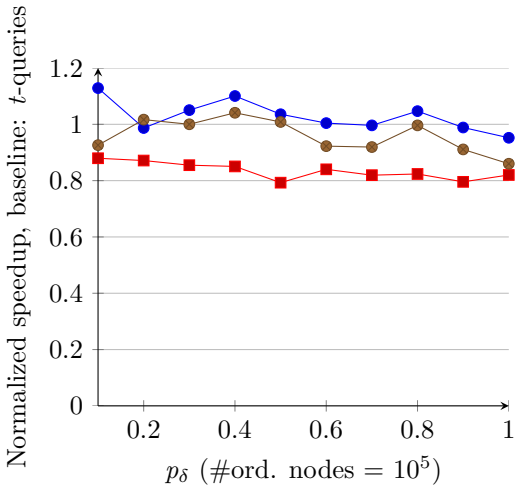
(b) Results for category 2 (Prec/Foll Sibling)



(c) Results for category 3 (Anc/Desc)

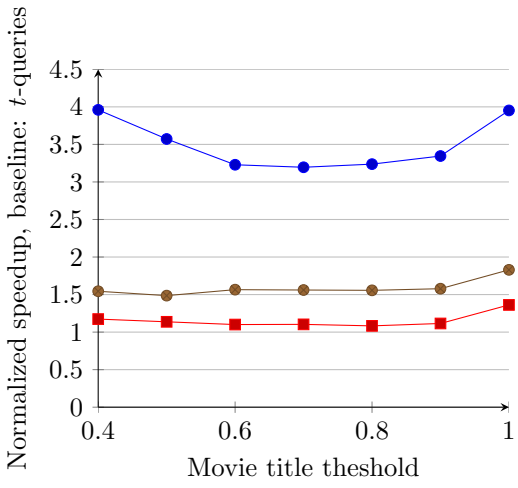


(d) Results for category 4 (Par/Child)

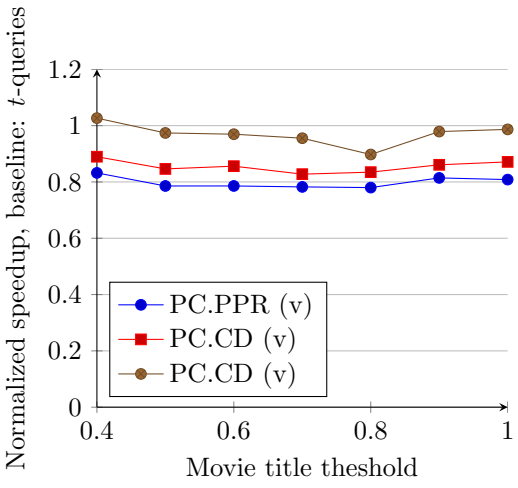


(e) Results for category 5 (Simple)

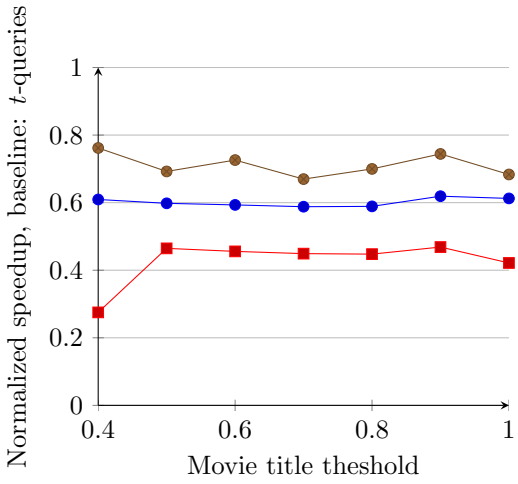
Figure C.7: Experimental results for increasing amounts of uncertainty



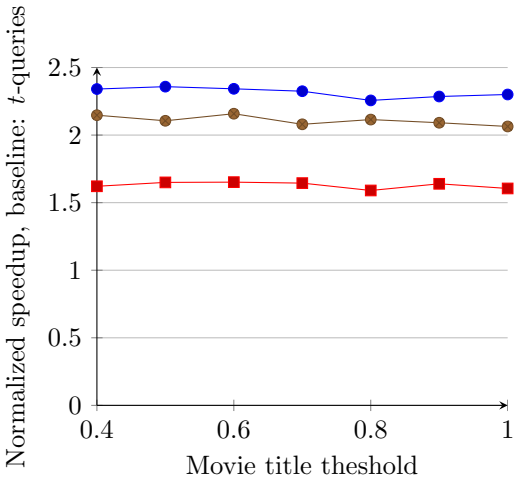
(a) Results for category 1 (Complex)



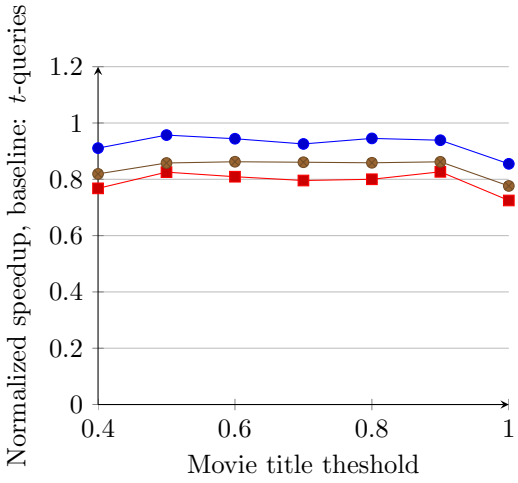
(b) Results for category 2 (Prec/Foll Sibling)



(c) Results for category 3 (Anc/Desc)



(d) Results for category 4 (Child/Parent)



(e) Results for category 5 (Simple)

Figure C.8: Experimental results for real world uncertain test set

Appendix D

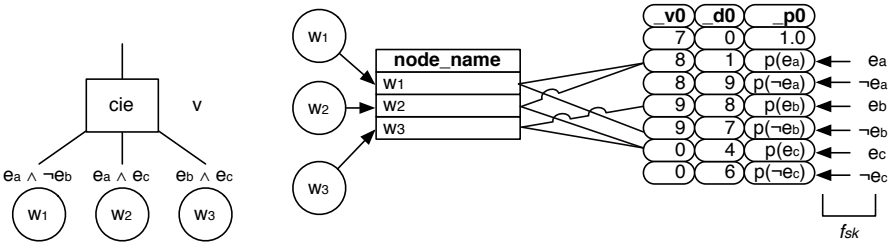
Retrospect on expressiveness

The approach taken in this thesis to map P-XML data into a URDBMS is based on the P-XML data model by Van Keulen & De Keijzer [50]. This data model is member of the $\mathsf{PrXML}^{\{\mathsf{ind,mux}\}}$ family. According to Kimelfeld [33], less expressive data models can be expressed in more expressive data models. As a consequence, our approach is applicable to P-XML data models member of $\mathsf{PrXML}^{\{\mathsf{ind,mux}\}}$ or less expressive P-XML families.

Kimelfeld et al. [33] state that both the $\mathsf{PrXML}^{\{\mathsf{cie}\}}$ family and the $\mathsf{PrXML}^{\{\mathsf{exp}\}}$ family are more expressive than the $\mathsf{PrXML}^{\{\mathsf{ind,mux}\}}$ family. This raised the question if a mapping of a more expressive probabilistic XML family into MayBMS is also feasible. We only consider the $\mathsf{PrXML}^{\{\mathsf{cie}\}}$ family in the following part of this section, since we believe that $\mathsf{PrXML}^{\{\mathsf{exp}\}}$ is not suitable to map with our approach.

The $\mathsf{PrXML}^{\{\mathsf{cie}\}}$ family uses distributional nodes of the *cie* type. These nodes are associated with independent random boolean variables $e_1, \ldots, e_m$ which are called events. Each event has a certain probability $p(e_i)$ of being 'true'. Events can be used by different cie nodes. A cie node v specifies for each child w a conjunction $\alpha^v(w) = a_1 \wedge \cdots \wedge a_n$ where each a_j is either e_i or $\neg e_i$ for $1 \leq i \leq m$. Node w exists if its corresponding conjunction $\alpha^v(w)$ is true.

We make the mapping of elements of node kind *cie* to U-Rel plausible by example. Figure D.1a shows an example of a cie node v with children w_1, w_2, w_3. The world set of w_1 is described by the events e_a and $\neg e_b$. Analogously, the world set of w_2 is described by the events e_a and e_c and the world set of w_3 is described by the events e_b and e_c. An event is either true or false. Thus, a set of events is basically a set of choices for which each choice has two alternatives. Mapping f_{sk} –introduced in Section 8.3– maps choices with an unlimited finite number of alternatives to a set of RVAs. We apply f_{sk} to the events in Figure D.1a that we interpret as a set of choices in order to obtain the left most U-Relation in Figure D.1b. Additionally, we map nodes w_1, w_2, w_3 to a set of rows. Finally, relate these rows to RVAs such that the world set of w_1, w_2, w_3 is described with RVAs.



(a) Input P-XML data

(b) Output relations

Figure D.1: Sketch of a more expressive P-XML into URDBMS data mapping

Acronyms

f_{fl}
: Flesh mapping. 7, 8, 76, 77, 110, 121, 123, 128, 133–135

f_{rk}
: Repair-key statement. 103–106, 108, 135, 136

f_{sk}
: Skeleton mapping. 7, 8, 76, 77, 103, 110, 121, 123, 128, 133–135, 157

ASI[*ENC*]
: Abstract Shared Inlining. 16, 74

ASI[*XA*]
: Abstract Shared Inlining with XA as parameter. 73–76, 85, 94, 95, 110, 111, 131, 134, 136

BB
: Batch based application. 83, 84, 87–89, 91, 93, 111, 112, 116, 118, 120, 121, 123–127, 129, 136

CD
: Glue by closest dependency. 31, 88, 89, 97, 112, 113, 118–129, 152–156

CPA
: Choice point assignment. 26, 27, 53, 57, 58, 60, 61, 72, 73, 76, 163

D
: Glue by depth. 92, 93, 112, 118–121, 131

DO
: Document oriented gluing. 94, 95, 110, 111, 113, 116, 121, 122, 128, 133–136, 161

DTD
: Document Type Definition. 8, 15, 16

P-XML
: Probabilistic XML. 1, 3–10, 21–23, 25–27, 32, 37, 50, 53–60, 62, 63, 67, 71–73, 76, 86, 94, 100, 108, 111, 113, 118, 121, 122, 131, 133–136, 157, 161, 162, 164

PB
: Partition based application. 84, 118, 120–123, 128, 131, 133, 135

PC
: Precomputed chaining. 31, 84, 85, 92, 118, 120–130, 133, 135, 136, 152–156

PPR
: Glue by possibility parent reference. 31, 88–90, 111, 118–130, 133, 135, 136, 152–156

QRO
: Query result oriented gluing. 94, 96, 113, 117, 125, 127, 129, 133–135, 161

RDBMS
: Relational database management system. 6, 8, 9, 13, 15, 16, 18, 19, 43, 44, 94, 103, 134, 136, 164–167

RVA
Random variable assignment. 27, 29, 31, 44, 49, 50, 53, 61, 64, 65, 72, 73, 76–81, 83–89, 91–94, 96–100, 103, 105, 110–112, 135, 157, 161–164

SI
Shared Inlining. 8, 13, 15, 16, 18, 73, 74, 131, 134, 161, 163

SW
Glue by sandwich. 89, 90, 97, 111–113, 118–121

UoC
Unit of Choice. 21

URDBMS
Uncertain relational database management system. 3, 4, 6–10, 21, 32, 37, 44, 63, 67, 71–73, 76, 78, 79, 84, 86, 94–97, 99, 103, 108, 113, 116, 118, 120–122, 131, 133–136, 157, 161, 164

WSD
World set descriptor. 23, 44, 45, 47, 49–51, 55, 58, 61, 63–65, 77–81, 95, 100, 110, 166

XA
XPath Accelerator. 9, 13, 15, 18, 19, 29, 73–76, 78, 94, 110, 111, 131, 134, 136, 141, 159

XML
eXtensible Markup Language. 3, 4, 6, 9, 13–16, 18, 19, 21–23, 26, 59, 94, 134, 136, 167

Glossary

$\mathcal{R}$
: An abstract formalism of a traditional data model that consists of the concepts *Database*, *Query* and *Match*. 6, 37, 43, 44, 164

$\mathcal{U}$
: An abstract formalism of an uncertain data model that consists of the concepts *U-Database*, *Query* and *U-Match*. 6, 7, 9, 10, 27–31, 37, 45, 50, 51, 53, 60–64, 67, 68, 73, 134, 164, 165

***t*-query**
: An SQL query derived from an XPath to SQL query mapping. A P-XML into URDBMS data mapping that includes a DO glue process allows for the evaluation of *t*-queries. 7, 95, 113, 121, 122, 124–130, 133, 135, 136, 147, 152, 154

***tg*-query**
: An SQL query derived from an XPath to SQL query mapping combined with a QRO glue process. A P-XML into URDBMS data mapping that does not include a DO glue process allows for the evaluation of *tg*-queries. 7, 8, 96, 97, 113, 121, 122, 124, 126, 128, 130, 133, 135, 136, 147, 152, 154

Abstract Shared Inlining
: Equal to the SI approach, except that the representation of nodes in a relational table is not yet defined. 16, 73–76, 85, 94, 95, 110, 111, 131, 134, 136, 159

alternative selection
: A probability node that selects a child possibility node as its one and only child (also referred to as a choice). 54–57

axis
: A tree-relationship between nodes in a tree. 14

batch based application
: A glue method application designed to glue one RVA to each entry of an input phase table per phase. 83, 84, 87–89, 91, 93, 111, 112, 116, 118, 120, 121, 123–127, 129, 136, 159

c-document
: A document instance of the C-XML data model. 26, 27, 53, 58–60, 62, 64, 71

C-XML
: A data encoding for documents that represents uncertain information as a tree. 26, 27, 32, 53, 54, 57–61, 71, 161

candidate
: specifies a required object. Candidates are used to specify a pattern. 29, 31

choice point assignment
: An assignment of a possibility node to a probability node that represents a viable dependency. 26, 27, 53, 57, 58, 60, 61, 72, 73, 76, 159, 163

commutative property
: A diagram commutes if all directed paths with the same start and end point lead to the same result by function composition. 13, 15, 25

conditional column
: Columns that hold either random variable identifiers, assignment values or probabilities of rows in a U-Relation. 31

context node
A node in a tree that is the starting point of a tree traversal. 15

data mapping
An operation that maps an instance of one data structure to an instance of another data structure. 6, 13, 32, 113

data model
A query language and a data structure for which query evaluation is defined. 4, 9, 37, 43, 44, 50, 53, 136

database mapping
A combination of a data mapping with corresponding query mapping. 4, 6–10, 13, 25, 32, 50, 53, 113, 133–135

dependency handle
Information that describes which entries in the skeleton are assigned to the phase table entry with which the dependency handle is associated.. 78, 87–90, 92, 94

depth
The maximum number of viable dependencies assigned to an object –node, row, table, p-document, query. 77, 90, 92, 118, 120

distributional node
A virtual node that specifies how children are randomly selected in a p-document. In this work, distributional nodes refer to possibility nodes and probability nodes. 7, 8, 10, 22, 53, 54, 57, 58, 71, 72, 76, 86, 108, 122, 133, 135, 165

document oriented gluing
An application of a glue process to the flesh of a p-document such that the glue process is part of the data mapping. 94, 95, 110, 111, 113, 116, 121, 122, 128, 133–136, 159, 161

don't care choice
A choice point with only one alternative is added to a p-document during a P-XML to U-Rel mapping. 77–79, 162

don't care RVA
An RVA that represents the don't care choice in a p-document. 79, 84, 85, 87, 91, 93, 94, 112

element table
A table created by an XML into RDBMS mapping that stores all nodes of a specific element kind (or multiple element kinds). 16, 116, 118, 120, 163, 166

element type
A property of an XML-node specified by its tag. 14–19, 163–166

eXtensible Markup Language
A data encoding for documents that represents information as a tree. 3, 4, 6, 9, 13–16, 18, 19, 21–23, 26, 59, 94, 134, 136, 160, 167

flesh
A tree that contains all ordinary nodes in a p-document. 7, 8, 10, 71, 72, 76, 77, 83, 85–88, 91, 110, 113, 116, 131, 133

flesh driven glue method
A glue method that takes the perspective of an ordinary node that is searching for all its ancestor possibility nodes. 85, 86

flesh result
The result of a query evaluated on the flesh. 96, 117

G-Join
A special statement that appends RVAs to a U-Relation. 77–79, 81, 87, 89, 90

glue by closest dependency
A flesh driven glue method that matches flesh entries to skeleton entries by finding the closest viable dependency of each flesh entry. 31, 88, 89, 97, 112, 113, 118–129, 152–156, 159

glue by depth
A skeleton driven glue method that matches flesh entries to skeleton entries by finding for each skeleton entry of a certain depth all its descendants. 92, 93, 112, 118–121, 131, 159

glue by possibility parent reference
A flesh driven glue method that matches flesh entries to skeleton entries with a possibility parent reference of flesh entries. 31, 88–90, 111, 118–130, 133, 135, 136, 152–156, 159

glue by sandwich
A flesh driven glue method that matches flesh entries to skeleton entries by finding for each flesh entry the one ancestor possibility node that does not have a descendant possibility node that is ancestor of the processed flesh entry. 89, 90, 97, 111–113, 118–121, 160

glue method
A function that matches rows to RVAs or nodes to CPAs. 78, 83, 85, 113, 118, 120, 121, 162, 163, 166

glue method application
The order in which RVAs are associated to rows or CPAs are associated to nodes. 83, 163

glue process
A step-wise approach that reconstructs the notion of a world set for data entries. A glue process is defined as a combination of a glue method with a glue method application. 10, 71, 77, 78, 80, 83, 113, 116–118, 120–122, 124, 126, 133, 135

guest/host-relationship
A relationship between a guest element type and a host element type such that the guest element type visits the host element type. 17, 18

identity function
A function that returns the same value as its input argument. 60

inlining
A part of the SI approach that allows nodes of different element types to be stored in the same element table if some requirements, specified by a set of inline rules, are satisfied. 16, 17

location step
An XPath operation –like ancestor, parent, sibling– that takes a set of context nodes as input and returns a set of nodes. 15, 164

master element type
The only element type in an element table that is not inlined with any other element type. 18, 76

MayBMS
A URDBMS built on top of PostgreSQL that stores data in U-Relations. 8, 37, 43–45, 47, 51, 79, 103, 106, 113, 135, 166

multi-union approach
A generic applicable approach optimizes SQL queries that contain n-ary union operations. 103

node test
A filter that restricts the result of a location step to contain solely elements of the element type specified in the node test. 15, 29

normalized speedup
A measure of how many times an algorithm a is faster than another algorithm a' based on normalized execution time. 117, 122, 124, 126, 128, 130

ordinary node
A node that is either an XML node or a text node. 7, 10, 14, 26, 71, 76, 85, 86, 90, 110, 122, 133–135, 162

p-document
A document instance of P-XML. 4–8, 10, 14, 22, 23, 26, 53–56, 58, 59, 61, 62, 71–73, 76, 79–81, 85, 86, 113, 114, 131, 133, 134, 141, 162, 164–166

partition based application
A glue method application designed to glue all RVAs to the entries of one partition of a an input phase table per phase. 84, 118, 120–123, 128, 131, 133, 135, 159

pattern
A query specified for $\mathcal{U}$ (or $\mathcal{R}$) that is constructed as a graph. 29, 161

phase
A step in a glue process that reconstructs part of the viable dependencies. 77, 83, 91, 161, 164, 165

phase table
The result of a phase. 77, 78, 83, 85, 116, 161, 162, 164, 165

possibility node
A distributional node that represents one alternative of a choice point. 22, 27, 54, 57, 72, 85, 86, 90, 120, 135, 162, 164

possibility parent
A possibility node that is reached with the smallest amount of parent location steps. 86

possible answer
The result of a query evaluated on one possible world is a possible answer. 23

possible document
A p-document where each probability node has exactly one child possibility node. 22, 54, 57

possible worlds model
A model on which uncertain databases are founded. 21, 23, 166

PostgreSQL
An open source RDBMS. 43, 103, 113, 116, 117, 164

pre-order
The process of ranking nodes in a tree such that for each node holds that its rank is smaller than nodes in its child axis and sibling axis. A second definition: the pre-order ranks nodes in a tree in the order in which they are discovered with a counter-clockwise traversal of the perimeter of that tree starting at the root. 18, 167

precomputed chaining
A glue method application designed to glue one skeleton path –represented as a set of RVAs– to each entry of an input phase table per phase. 31, 84, 85, 92, 118, 120–130, 133, 135, 136, 152–156, 159

probabilistic XML
A data encoding for documents that represents uncertain information as a tree. 1, 3–10, 21–23, 25–27, 32, 37, 50, 53–60, 62, 63, 67, 71–73, 76, 86, 94, 100, 108, 111, 113, 118, 121, 122, 131, 133–136, 157, 159, 161, 162, 164

probability node
A distributional node that represents a choice point. 22, 27, 54, 57, 90, 114, 162, 164

query mapping
An operation that maps queries specified in one query language to semantically equivalent queries specified in another query language. 13, 32, 113

query planner
A component of an RDBMS that attempts to determine the most efficient approach to evaluate a query described with a query execution plan. 117, 120, 122, 124, 126

query result oriented gluing
An administering of the glue process to query results such that the glue process itself is part of the query execution process.. 94, 96, 113, 117, 125, 127, 129, 133–135, 159, 161

random document
the XML-document that is left over after distributional nodes are discarded from a possible document. 22, 23, 55

random variable assignment
An assignment of a value to a random variable that represents a viable dependency. RVAs are used by $\mathcal{U}$ and U-Rel to represent mutual exclusive choices. 27, 29, 31, 44, 49, 50, 53, 61, 64, 65, 72, 73, 76–81, 83–89, 91–94, 96–100, 103, 105, 110–112, 135, 157, 160–164

relational database management system
A table-structured database management system based on relational algebra. 6, 8, 9, 13, 15, 16, 18, 19, 43, 44, 94, 103, 134, 136, 159, 164–167

repair-key statement
A statement that transforms an inconsistent certain table into a consistent uncertain table (a U-Relation). 103–106, 108, 135, 136, 159

Shared Inlining
A schema-based XML into RDBMS mapping that stores XML nodes of the same element type in the same relational table. 8, 13, 15, 16, 18, 73, 74, 131, 134, 160, 161, 163

skeleton
A tree that contains all distributional nodes in a p-document. 7, 10, 71, 72, 76–78, 87, 88, 90–92, 103, 110, 113, 131, 133, 162

skeleton driven glue method
A glue method that takes the perspective of a possibility node that is searching for all its descendants. 85, 86, 90

skeleton path
The set of possible choices that capture the aliveness of nodes in a p-document. 57, 84, 85, 98, 165, 166

skeleton path table
A U-Relation that contains all skeleton paths of a certain maximum size. 84, 85, 113, 122

slave element type
An element type in an element table that is inlinined with some other element type. 18, 76

speedup
A measure of how many times an algorithm a is faster than another algorithm a' based on real execution time. 108, 117, 122

twig pattern
An alternative representation of an XPath expression specified as a tree. 27, 29

U-Rel
Short notation for the U-Relational data model –the data model of MayBMS. 4–8, 10, 25, 27, 29, 44, 50, 61, 63, 64, 71, 76, 79, 95, 134, 157, 162, 165

U-Relation
An uncertain relational table for which a WSD is assigned to each row. 4, 6, 31, 44, 50, 63–65, 71, 77–79, 81, 83, 95, 100, 103, 118, 133, 157, 161, 164, 166

uncertain database
A database that stores a representation of a set of possible worlds. 13, 21, 23, 166

uncertain relational database management system
An RDBMS that integrates the possible worlds model. 3, 4, 6–10, 21, 32, 37, 44, 63, 67, 71–73, 76, 78, 79, 84, 86, 94–97, 99, 103, 108, 113, 116, 118, 120–122, 131, 133–136, 157, 160, 161, 164

uncertainty distribution
Information that describes how database objects are randomly selected in an uncertain database. 21, 31, 53, 133

uncertainty management mechanism
A mechanism that transforms an ordinary database management system into an uncertain database management system such that (1) a database represents a set of possible worlds, and (2) query evaluation adheres to the possible world semantics. 6, 37, 43, 51, 103

uncertainty ratio
A property of a p-document calculated as the number of ordinary nodes divided by the number of possibility nodes. 114, 115

Unit of Choice
Refers to the granularity of uncertainty in an uncertain database. 21, 160

viable dependency
An element y is the viable dependency of another element x if in each possible world of which x is member, y is also member of that possible world. 77, 84–86

view
A virtual table representing a select query. 23, 117

world set
The set of possible worlds of which an object is member. 21, 23, 26, 27, 78, 163, 167

world set descriptor
Information that describes a set of possible worlds. The world set descriptor of an object, row or node describes its world set. 23, 44, 45, 47, 49–51, 55, 58, 61, 63–65, 77–81, 95, 100, 110, 160, 166

XPath
Query language to specify tree traversals in XML-documents. 3–5, 7–10, 13–15, 18, 19, 27, 32, 60, 62, 67, 75, 100, 113, 115, 116, 118, 122, 131, 133–135, 163, 166

XPath Accelerator
A schema-less XML into RDBMS mapping that encodes a tree with the pre-order, size and level of nodes in that tree. 9, 13, 15, 18, 19, 29, 73–76, 78, 94, 110, 111, 131, 134, 136, 141, 159, 160