

REASONING ABOUT FUNCTIONAL AND KEY
DEPENDENCIES IN HIERARCHICALLY STRUCTURED DATA

CARMEM SATIE HARA

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy.

2004

Susan Davidson
Supervisor of Dissertation

Wenfei Fan
Supervisor of Dissertation

Benjamin Pierce
Graduate Group Chairperson

To my parents.

To Wagner and Pedro.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors, Susan Davidson and Wenfei Fan for their guidance and advice. I have no doubt I would not have succeeded in concluding this dissertation without Susan's constant reassurance and her enormous generosity. Her insights, knowledge, and experience were crucial in the definition of goals and directions, and in the development of the dissertation. Wenfei provided guidance through most of the theoretical work. I will be forever grateful to him for helping me see beauty in a once mysterious field of database theory.

My heartfelt thanks go to the committee members for their insightful comments and suggestions: Val Tannen, Peter Buneman, and Insup Lee. I would also like to thank Peter Buneman, the *father* of Keys for XML, for introducing me to the concept on which the main results of the dissertation were developed. Jing Qin did much of the work on the experimental part described in Chapter 6. I feel privileged for having been a member of the Penn Database Group within which I had spent the most intellectually challenging years of my life. Some of my contemporaries in the group, that greatly influenced this work are: Peter Buneman, Yi Chen, Byron Choi, Jonathan Crabtree, Susan Davidson, Alin Deutsch, Wenfei Fan, Scott Harker, Kyle Hart, Anthony Kosky, Zoé Lacroix, Hartmut Liefke, Rona Machlin, Lucian Popa, Arnaud Sahuguet, Wang-Chiew Tan, Val Tannen, and Yifeng Zheng.

I would also like to thank the administrative staff of the CIS Department. A very special thanks go to Mike Felker, the guardian angel of all graduate students, who really looked over me from my first year at Penn to this day. There are also many people who helped in making my time at Penn enjoyable, and I cannot refrain from mentioning a few: Karin Kipper, William Schuler, Emilio Del Moral Hernandez, Siome Goldstein, and Carlos Prolo, who so kindly offered me a place to stay during my visits to Penn after I moved back to Brazil.

I would be remiss if I did not mention my gratitude to my mentors during the master program years at Unicamp, Brazil: Geovane Cayres Magalhães and Tadao Takahashi. Geovane supervised my master thesis, and Tadao supported my first experience abroad

at UCLA, under the supervision of Gerald Estrin. Professor Estrin gave the Brazilian troop such a warm welcome that inspired me to pursue the doctorate in the United States. Pursuing the doctorate abroad would not be possible without the support from CNPq (Brazilian Council for Scientific and Technological Development) and from UFPR (Federal University of Parana), for which I am especially grateful. I would also like to thank my colleagues at UFPR who had given me encouragement and support during my years at Penn.

Finally, but most importantly of all, I would like to thank my parents, brothers, and sisters for their love. My parents taught me the value of education, and they wished that all their children would become “doctors” one day. I am really happy to be the last one to obtain the title and finally fulfilling their dream. I could not have achieved any of this without the love, care, and enormous amount of patience of my husband, Wagner Zola. We have started this journey together, and now we have a third companion, Pedro, who brought such a joy to our lives, we cannot imagine life without him.

ABSTRACT

**REASONING ABOUT FUNCTIONAL AND KEY DEPENDENCIES
IN HIERARCHICALLY STRUCTURED DATA**

Carmem Satie Hara

Supervisors: Susan Davidson and Wenfei Fan

This dissertation investigates how constraints can be used to check the consistency of data being exchanged between different sources. Data exchange involves transformations of data, and therefore the “transformed” data can be seen as a view of its source. Thus, the problem we investigate is how constraints are propagated to views, when the data involved is not restricted to relational tables, but may be hierarchically structured in several levels of nesting. The ability to determine constraint propagation relies on the ability to determine constraint implication. This is because the validity of a constraint on the view may not result directly from constraints defined on the source data, but from their consequences. Therefore, the dissertation starts by investigating two forms of constraints: nested functional dependencies and keys for XML, and their implication problems. More specifically, we present a definition of functional dependencies for a nested relational model, and a sound and complete set of inference rules for determining logical implication for the case when no empty sets are present. Motivated by the popularity of XML as a data exchange format, we present a definition of keys for XML that are independent of any type specification. We study two notions of keys: *strong keys*, and *weak keys*, and for each of them we derive a sound and complete set of inference rules, as well as algorithms for determining their implication. Capitalizing on the results of XML key implication, we investigate the problem of propagating XML keys to relational views. That is, the problem of determining what are the functional dependencies that are guaranteed to hold in a relational representation of XML data, given that a set of XML keys hold on the XML document. We provide two algorithms: one is to check whether a functional dependency is propagated from XML keys, and the other is to compute a minimum cover for all functional dependencies on a universal relation given certain XML keys. The ability to compute XML key propagation is a first step toward establishing a connection between XML data and its relational representation at the semantic level.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Contributions	7
1.3	Organization	11
2	Functional Dependencies for a Nested Relational Model	14
2.1	Functional Dependencies	17
2.1.1	Data Model	17
2.1.2	Path Expressions	19
2.1.3	Nested Functional Dependencies	21
2.1.4	NFDs expressed in logic	24
2.2	Discussion	26
3	Reasoning about Nested Functional Dependencies	29
3.1	Axiomatization for NFD Implication	30
3.1.1	Completeness of the NFD-rules	35
3.2	The Problem of Empty Sets	47

3.3	Simple NFDs	50
3.3.1	Join and Multivalued Dependencies (MVDs)	50
3.3.2	Construction of the Flattened Representation of a Nested Relation	55
3.3.3	Relationship between NFDs and FDs+MVDs	61
3.4	Discussion	64
4	Keys for XML	67
4.1	Keys	71
4.1.1	A Tree Model and Value Equality	71
4.1.2	Path Languages	72
4.1.3	A Key Constraint Language for XML	75
4.1.4	Transitive Set of Keys	78
4.2	Discussion	79
4.2.1	XML-Schema	80
4.2.2	Strong Keys	81
4.2.3	Keys that Determine Value Equality	83
4.3	Comparison with Nested Functional Dependencies	83
4.4	Functional Dependencies for XML	87
5	Reasoning about Keys for XML	90
5.1	Decision Problems	91
5.2	Key Implication	93
5.2.1	Inclusion of <i>PL</i> Expressions	94

5.2.2	Axiomatization for Absolute Key Implication	101
5.2.3	Axiomatization for Key Implication	107
5.2.4	Axiomatization for Strong Key Implication	122
5.3	Discussion	126
6	Propagating XML Constraints to Relations	128
6.1	Transformations from XML to Relations	132
6.2	Problem Statement and Limitations	136
6.2.1	Key propagation	136
6.2.2	Minimum cover	139
6.2.3	Propagation of other XML constraints	140
6.3	Checking Key Propagation	143
6.3.1	Propagation Algorithm	144
6.3.2	The Correctness of the Propagation Algorithm	150
6.4	Computing Minimum Cover	161
6.4.1	A Naive Algorithm	161
6.4.2	A Polynomial-Time Algorithm	162
6.4.3	The Correctness of the Minimum Cover Algorithm	175
6.5	Experimental Study	188
6.5.1	Experimental Testbed	189
6.5.2	Experimental Results	189
6.6	Discussion	194

7 Conclusion	196
7.1 Contributions	196
7.2 Further Work	200

List of Tables

3.1	Armstrong Axioms for FD implication	30
3.2	Rules for NFD implication	31
3.3	Rules for FD+MVD implication	52
5.1	\mathcal{I}_p : Rules for PL expression inclusion	95
5.2	\mathcal{I}_{abs} : Rules for absolute key implication	102
5.3	\mathcal{I} : Rules for key implication	106
5.4	\mathcal{I}_{att} : Rules for strong key implication	123

List of Figures

1.1	XML data represented as a tree	9
2.1	An instance that violates $R[B : C \rightarrow E : F]$	24
3.1	Algorithm for building an instance I of $Sc(R)$	38
3.2	A relation that satisfies $\bowtie [AB, AC, BD]$	51
3.3	A tableau query (T_σ, t) , where $\sigma = \bowtie [AB, AC, BD]$	53
3.4	Application of a jd-rule	54
3.5	A schema tree and its extension with set-identifier attributes	55
3.6	A nested relation extended with set-identifier attributes	57
3.7	Flattened representation of a nested relation	57
3.8	Relation that satisfies $Id_x \twoheadrightarrow y_1, y_k, Id_x \rightarrow y_1, Id_x \rightarrow y_k$, but not $y_1, y_k \rightarrow Id_x$	63
4.1	Example of some XML data and its representation as a tree	69
4.2	Illustration of a key $(Q, (Q', \{P_1, \dots, P_k\}))$	76
4.3	Data represented as a nested relation and as an XML tree	84
4.4	DTD of the document in Figure 4.3	85
4.5	Interaction between DTDs and XML keys	86
4.6	Enrollment document	88

5.1	NFA for the PL expression $a//a/c//b$	96
5.2	Algorithm for testing inclusion of PL expressions	100
5.3	Finite implication of absolute keys	103
5.4	Abstract trees constructed in the proof of Lemma 5.6	114
5.5	Finite implication of \mathcal{K} constraints	120
5.6	Finite implication of \mathcal{K}_{att} constraints	125
6.1	Tree representation of XML data	129
6.2	Sample relational instances	130
6.3	Table trees	134
6.4	Instances generated by the transformation of Example 6.1	135
6.5	Table trees	141
6.6	Table trees to illustrate key propagation	146
6.7	An algorithm for checking XML key propagation	148
6.8	A naive algorithm for finding a minimum cover	162
6.9	Trees representing universal relations	163
6.10	Computing minimum cover	167
6.11	Procedure <code>computeKeys</code>	168
6.12	Minimization of FDs	169
6.13	Procedure <code>genFDs</code>	171
6.14	Time for computing minimum cover in seconds	190
6.15	Effect of depth of the table tree on the time for computing XML key prop- agation	192

6.16 Effect of number of keys on the time for computing XML key propagation . 193

Chapter 1

Introduction

This dissertation investigates key and functional dependencies in hierarchically structured data and their implication problems. More specifically, we will present a definition of functional dependencies for a nested relational model, and a definition of keys for XML, and show that they are finitely satisfiable and moreover, that there exists a sound and complete set of inference rules for determining their implication.

There is a natural analogy between this work and the theory of functional dependencies for the relational model. The theory of functional dependencies constitutes an important part of the relational database theory. It forms the basis of the normalization theory for the relational model, and it is also useful for query optimization and to study how dependencies are carried from a database to a view. In all these applications, it is important to be able to reason about functional dependencies. If we are to achieve the same functionality for both the nested relational model, and for XML, it is important to study the implication problem of dependencies in these new settings.

In fact, capitalizing on the results of XML key implication, we investigate the problem of propagating XML keys to relational views of XML data. That is, the problem of determining what are the functional dependencies that are guaranteed to hold in a relational representation of XML data, given that a set of XML keys hold on the XML document. In the dissertation, we provide two algorithms for computing XML key propagation. One

algorithm is to check whether a functional dependency is propagated from XML keys via a predefined view, and the other is to compute a minimum cover for all functional dependencies on a universal relation given certain XML keys.

1.1 Motivation

The main motivation for studying data dependencies is to incorporate more semantics into data models, including those with richer constructs than the relational model, and that allow data to be deeply nested. One of the contexts in which data dependencies play an important role is when data is being exchanged between different sources. A common paradigm in many application areas is for a data provider to export its data using a data exchange format; on the other end, the data consumer imports some or all of the data and stores it using database technology. Clearly, the mapping must be guided by an understanding of the semantics of the data. The problem arises when exporters and importers have different such understandings, and when the semantics cannot be captured by the schema definition alone. This makes it very difficult to write the transformations and to reason about their correctness.

It would be helpful if tools existed to facilitate the general problem of mapping between different data formats, taking the semantics of data into account. To facilitate such mapping, we need a language in which to express transformations and constraints, and the ability to reason about the correctness of the transformations with respect to the constraints.

Over the past five years, XML has become enormously popular as a data exchange format. The appeal of XML is that it is a way of serving data in a uniform, flexible, and easily parsable form. Data is self-describing and hence easier to understand, and there are many freely available parsers and other tools for XML. Moreover, XML data does not require the existence of any type specification, in the form of a DTD[Lay98a] or XML Schema[TBMM01]. Even if a type specification exists, it does not solve the data exchange problem by itself. Exporters must *map* their data into the exchange format, and importers of data must again map from the exchange format into their local format. Thus data

exchange is inextricably tied up with writing mappings (or *transformations*) between data formats. In fact, over the past fifteen years, much attention has been paid to the problem of developing query and transformation languages for complex and object-oriented data sources [DK97, BLS⁺94, DHP97, LDB97, DCB⁺01].

In bioinformatics, one scenario in which these problems arises is in the context of the Microarray Gene Expression (MAGE) standard[ER01]. The semantics of MAGE is specified using the Universal Modeling Language (UML) (the “object model”, MAGE-OM). This standard is then expressed as an XML DTD for data exchange (MAGE-ML). Prior to the MAGE effort, a relational database called the RNA Abundance Database (RAD) had been developed at the Penn Center for Bioinformatics to store gene expression data as well as its associated sample annotation data. Given the MAGE-ML standard, data is imported from collaborators and exported from RAD using this format. However, each of these data representations – RAD and MAGE-OM/ML – have been developed independently. Each of the data representations has an extensive schema, in which tables/entities have keys and possibly foreign keys, relationships have multiplicity and may be required, and attributes may have enumerated types. Data exchange between RAD and MAGE-ML will have to be validated as correct, not only in its mapping but with respect to the constraints expressed on the data in each representation. Two simple examples of the difficulties entailed are illustrated below:

1. The data exported by RAD into MAGE-ML through some transformation may fail to validate against the constraints of MAGE-OM. To provide compliant data, either data cleansing will have to be performed as the data is being exported into MAGE-ML, or RAD will have to be redesigned.

Example 1: In MAGE-ML the experimental samples from the laboratory for gene expression annotation are termed BioMaterials, and they can be of three types: BioSource, BioSample, and LabeledExtract. The BioSource is used to designate the innate (or starting) properties of a sample. BioSamples are derived from one or more BioSources (and/or BioSamples) through Treatment events, and the final BioMaterial, resulting from a sequence of Treatments, is the LabeledExtract [ea02].

This sequence of Treatments is represented in MAGE-ML as a directed acyclic graph, which starts from BioSources and ends with LabeledExtracts. In RADv2, the annotation interface required only information about the end result of the sample preparation, with an (optional) free-text description of the process. Since in MAGE-ML a BioSample and LabeledExtract can only exist if the BioSource is present, the data on experimental samples in RADv2 was inconsistent with MAGE-ML. RAD was therefore modified by extending the database schema. A new annotation interface was also developed to force the sample preparation process to be captured, and the required information to be stored.

2. The data imported by RAD through some transformation from MAGE-ML may violate integrity constraints in RAD. If the MAGE-ML data is consistent with respect to the constraints of MAGE-OM, then there must be some inconsistency between MAGE-OM and the constraints expressed in RAD. Thus, either RAD's schema will have to be re-designed to be consistent with MAGE-ML, or integrity checking will have to be turned off when data is imported.

Example 2: In the Experiment package of MAGE-ML, an Experiment represents the collection of results for one or more BioAssays. Each Experiment has a unique Identifier, a Name, and a ExperimentDesign, which can have many associated Types (e.g. “time course” and “normal vs. diseased”), and a single Description. In RADv2, there is a single relation `Groups(Group_ID, Group_Name, Group_Type, Group_Desc)` which corresponds to the Experiment class. The `Group_Id` is the key of the `Groups` relation, and is taken from the Identifier of the Experiment object. The `Group_Name` corresponds to the Experiment Name while `Group_Type` and `Group_Desc` correspond to the Type and Description of the ExperimentDesign, respectively. However, this is incorrect since there could be many different types associated with an Experiment rather than the single one implied by the key constraint in the relational design. Therefore, RADv2 had to be re-designed to correct this inconsistency.

The examples above identify two situations that have caused a re-design of RAD. They were caught by the programmer in charge of specifying the mapping from RAD to MAGE-ML. However are these the only problems that will be encountered as the process is completed?

The schemas involved are both extremely large, involving hundreds of attributes, making it very difficult to catch inconsistencies. Rather than recognizing inconsistencies through an ad-hoc process and laboriously going through successive redesigns of RAD to deal with them, it would be extremely helpful to have a framework in which, given a desired mapping of data and given existing constraints, all ensuing inconsistencies could be automatically exposed and corrections suggested.

Another scenario is that a group may wish to design a database to store data based on an XML exchange standard so that it can be easily exchanged. In this case, it would be very helpful to be able to specify the mapping between the exchange standard and some initial design of the database, have constraints on the database automatically be generated from the exchange standard via the mapping, and thus generate the database design automatically.

Even though XML has become a popular format for data exchange, a recognized problem with XML is that it is only syntax and does not carry the semantics of the data. To address this problem, a number of constraint specifications have recently been proposed for XML which include a notion of keys. A natural question to ask, therefore, is how information about constraints can be used to determine when an existing consumer database design is incompatible with the data being imported, or to generate de-novo a good consumer database. In this dissertation we present a framework for determining the set of functional dependencies that are guaranteed to hold on a relational representation of XML data, given that the XML document satisfies a set of keys. The ability to computing this set is a first step toward establishing a connection between XML data and its relational representation at the semantic level.

Revisiting Example 2, table **Groups** is populated from the XML data as follows: For each **Experiment** element, a tuple is created in the **Groups** relation containing the value of the **identifier** attribute for **Group_ID**, the value of **Name** for **Group_Name**, and from the **ExperimentDesign** subelement, the **type** value is extracted for **Group_Type**, and the **description** value is extracted for **Group_Desc**. The key of the **Groups** table has been specified as **Group_ID**.

It turns out that given the following keys on the XML data, the designers of RADv2 could prove that in the `Groups` relation it is indeed the case that `Group_ID` \rightarrow `Group_Desc`, `Group_Name`, but not `Group_ID` \rightarrow `Group_Type`, identifying therefore, a potential source of conflict:

1. `identifier` uniquely identifies a `Experiment` element.
2. Each experiment has a unique `name`, and a unique `ExperimentDesign`.
3. Within each `ExperimentDesign`, there exists a single `description`.

That is, if these XML keys hold on the data being imported, then `Group_ID` \rightarrow `Group_desc`, `Group_name` is a functional dependency (FD) that is guaranteed to hold on the `Groups` relation generated. We refer to the FD as one that is *propagated* from these XML keys.

The problem of determining whether an FD is propagated from a set of XML keys is closely related to the problem of determining view dependencies. If the underlying data model used is relational and the constraints considered are functional and multivalued dependencies, the problem has been well studied. That is, it is possible to derive the set of projected dependencies on a relational view computed using relational algebra, given dependencies on the base relations (see [Klu80, KP82, MMS79]). However, it is not even clear what “constraints” mean in a hierarchically structured model, and less clear as to what the inference rules are for reasoning about constraints. Reasoning about constraints is essential for determining view propagation, since a constraint defined on a view may not be determined directly by constraints defined on the base data, but by their consequences.

In this dissertation, therefore, we start by studying two forms of dependencies: functional dependencies for a nested relational model, and keys for XML. Then, capitalizing on the ability to reason about these dependencies efficiently, we will present algorithms to determine whether a set of functional dependencies are guaranteed to hold on a relational representation of XML data, given that the XML document satisfies a set of XML keys.

1.2 Contributions

This section gives an overview of the dissertation’s contents and summarizes our main contributions.

Nested Functional Dependencies In the *nested relational model* we adopt, record and set constructs can be nested by alternating. To illustrate this, consider the following nested relation schema *Course*, defined as a set of records with attributes *cnum*, *time*, and *students*, where *students* is a set of records with labels *sid*, *age*, and *grade*.

$$\textit{Course} : \{ \langle \textit{cnum}, \textit{time}, \\ \textit{students} : \{ \langle \textit{sid}, \textit{age}, \textit{grade} \rangle \} \rangle \}.$$

Some functional dependencies that we would like to be able to express for *Course* are:

1. *cnum* is a key.
2. In a given course, each student gets a single grade.
3. The assignment of *age* to a given student must be consistent throughout the *Course* relation.
4. A student cannot be enrolled in courses that overlap on time.

There are “local” dependencies, as dependency 2, where a student can have only one *grade* for a given course, but a different *grade* for distinct courses. There are also “global” dependencies as dependency 3, where the assignment of *age* to *sid* should be consistent throughout the *Course* relation. Dependency 4 illustrates how an attribute from an outer level of nesting is determined by attributes in a deeper level of nesting. Note that even if every level of nesting presents a “key”, as suggested in [AB86], this type of dependency is not captured by the structure of the data.

We introduce the notion of *nested functional dependencies* (NFDs), which are capable of expressing all these dependencies. NFDs are functional dependencies extended to allow simple path expressions. As an example, dependencies 2 and 3 above are expressed as:

$Course : [students : sid \rightarrow students : age]$

$Course : students : [sid \rightarrow grade]$

Having defined nested functional dependencies, we then turn our attention to the implication problem, that is, deciding if a given dependency is logically implied by a given set of dependencies. In the relational model, a sound and complete set of three axioms known as “Armstrong’s Axioms” suffice for this problem. Although reasoning with nested functional dependencies is strikingly more complicated, we have derived a sound and complete set of eight inference rules *in the case that empty sets are known not to occur anywhere*. Empty sets cause tremendous difficulties in reasoning since formulas such as

$$\forall x \in R.P(x)$$

are trivially true when R is empty. The additional complexity of dealing with empty sets has also been recognized in the context of query containment in [PT99] and [LS97].

Keys for XML

Several definitions of functional dependencies for XML (XFDs) have recently been proposed [CDHZ03, VLL04, AL04]. Although it would be tempting to study the implication of FDs for XML, we restrict our attention to keys, given the difficulties in reasoning about FDs. In fact, in [AL04], it is proven that their definition of XFDs is not finitely axiomatizable, and although a sound set of inference rules is presented for both the definitions proposed in [VLL04] and [CDHZ03], they are not proven to be complete. In contrast, our definition of keys for XML, which is an important special case of functional dependencies, can be reasoned about efficiently. Moreover, there exists a set of inference rules that are sound and complete for determining their implication.

To illustrate the type of dependencies that we would like to express, consider the XML document on the domain of books represented as a tree in Figure 1.1.

Some keys for this data might include:

1. A `book` node is identified by `@isbn`;

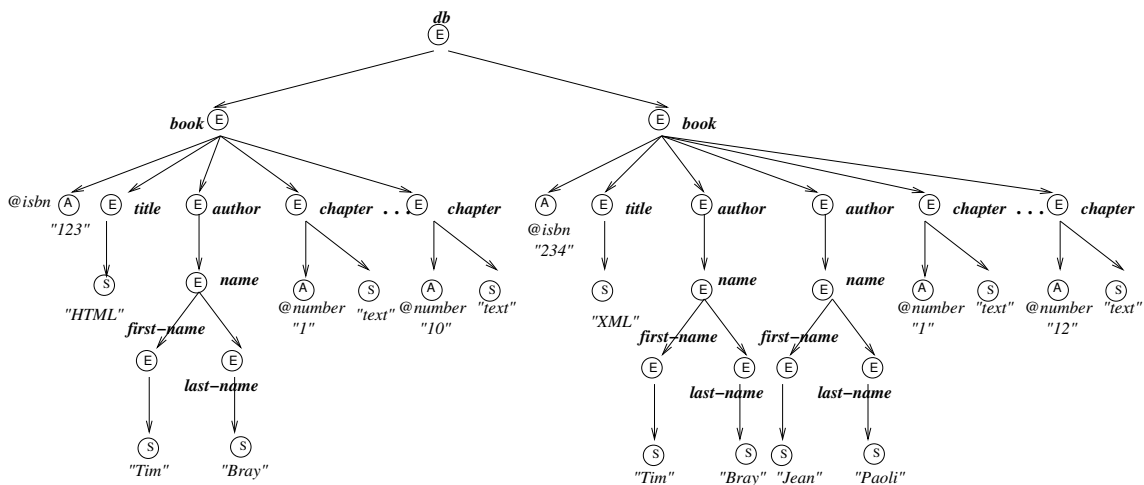


Figure 1.1: XML data represented as a tree

2. An `author` node is identified by `name`, no matter where the `author` node appears; and
3. Within any subtree rooted at `book`, a `chapter` node is identified by `@number`.

These keys are defined independently of any type specification. The first two are examples of “global” keys since they must hold globally throughout the tree. We denote them as *absolute keys*. Observe that `name` has a complex structure. As a consequence, checking whether two authors violate this constraint involves testing value-equality on the subtrees rooted at their `name` nodes. The last one is an example of a “local” (*relative*) key since it holds locally within each subtree rooted at a `book`. It should be noted that a `chapter @number` is not a key for the set of all `chapter` nodes in the document since two different books have chapters with `@number = 1`.

Key specifications for XML have been proposed in the XML standard [BPSM98], XML Data [Lay98b], and XML Schema [TBMM01]. However, existing proposals cannot handle one or more of the above situations. In particular, they are not capable of expressing the second and third constraints. To overcome these limitations, we propose [BDF⁺02] a new definition of keys for XML. As an example, dependencies 1 and 3 above are expressed as:

$$(book, \{ @isbn \}) \quad (book, (chapter, \{ @number \})).$$

Given the definition of keys for XML, we then turn our attention to their decision problems. We show that these keys are always (finitely) satisfiable, that is, given a set of keys, there exists a (finite) XML tree that satisfies the set. Moreover, there exists a set of inference rules that are sound and complete to determine their implication. Based on the rules, we provide a polynomial time algorithm for determining XML keys implication.

Propagation of XML constraints

Having determined that XML keys can be reasoned about efficiently, we present a framework for determining the set of FDs that are guaranteed to hold on a relational representation of XML data, given that the XML document satisfies a set of keys. It is worth remarking that the ability to compute such FDs depend on the ability to reason about XML keys. This is because some FDs may not be directly derivable from the XML keys defined on an XML document, but from their consequences.

As exemplified in the previous Section, this framework can be used for improving consumer relational database design. Our approach is based on inferring functional dependencies from XML keys through a given mapping (transformation) of XML data to relations. More specifically, we make the following contributions:

- A simple language that is capable of specifying transformations from XML data to relations of any predefined schema, and is independent of DTDs and other schema information for XML.
- A polynomial time algorithm for checking whether an FD on a predefined relational database is propagated from a set of XML keys via a transformation.
- A polynomial-time algorithm that, given a universal relation specified by a transformation rule and a set of XML keys, finds a minimum cover for all the functional dependencies mapped from XML keys.
- Undecidability results that show the difficulty of XML constraint propagation.
- Experimental results which show that the algorithms are efficient in practice.

Note that the polynomial-time algorithm for finding a minimal cover from a set of XML keys is rather surprising, since it is known that a related problem in the relational context

– finding a minimum cover for functional dependencies *embedded* in a subset of a relation schema – is inherently exponential [Got87].

The undecidability results give practical motivation for the restrictions adopted in our framework. In particular, one result shows that it is impossible to effectively propagate all forms of XML constraints supported by XML Schema, which include keys and foreign keys, even when the transformations are trivial. This motivates our restriction of constraints to a simple form of XML keys. Another undecidability result shows that when the transformation language is too rich, XML constraint propagation is also not feasible, even when only keys are considered. Since XML to relational transformations are subsumed by XML to XML transformations expressible in query languages such as XQuery [Cha01], this negative result applies to most popular XML query languages.

Although a number of relational storage techniques have been developed for XML [STZ⁺99, Sha01, Ora01, SKWW00, MFK⁺00, LC01], to the best of our knowledge, our framework and algorithms are the first results on mapping XML constraints through relational views. Being able to reason about constraints on views not only plays an important role in the design of relational storage of XML data, but is also useful for query optimization and data integration.

1.3 Organization

The remainder of this dissertation is organized as follows: Chapter 2 presents the definition of nested functional dependencies (NFDs). It naturally extends the definition of such dependencies for the relational model by using path expressions instead of attribute names. The meaning of NFDs is given by defining their translation to logic. We show that NFDs allow the definition of both local and global functional dependencies (intra and inter-set dependencies). They can also express some properties of sets. For example, it is possible to express that a given set is a singleton, and that sets do not share elements. We also discuss an alternative definition of NFDs, that we call simple NFDs. Simple NFDs have the same expressive power as NFDs. Although simpler in form, they are less intuitive because local

and global dependencies cannot be distinguished syntactically.

The inference rules for NFDs are the subject of Chapter 3. We present a set of eight rules that are sound and complete for the case where no empty sets are present. Conceptually, the rules can be broken up into three categories: three that mirror Armstrong's axioms, two that transform between NFDs and simple NFDs, and three that allow inferences based solely on the nested form of the data. The effects of the presence of empty sets on the definition of NFDs and the inference rules are also discussed. Moreover, we show that the set of inference rules for NFDs is not equivalent to the set of inference rules for functional and multi-valued dependencies applied to a flattened representation of a nested relation. An abstract of Chapters 2 and 3 was published in [HD99].

Chapter 4 presents the definition of keys for XML. Similar to the notion of NFDs, our definition of keys allows the definition of both local (relative) and global (absolute) keys, and it can also express that a given element can contain at most one value, that is, it is a singleton. Since a single relative key may not be able to uniquely identify a node in a XML tree, we introduce the notion of a transitive set of keys. That is, a set of keys that allows one to uniquely identify a node in the tree. We also discuss alternative forms of key definitions for XML. In particular, we present the notions of strong keys and weak keys. For strong keys, key values must exist and be unique, along the same lines as the definition of keys in the relational model. On the other hand, for weak keys, key values may not exist and may not be unique, that is, they may define a set of values. The motivation for this is to cope with the semi-structured nature of XML. A discussion on the similarities and differences between these keys and NFDs is also presented. Most of the material in this Chapter has been published in [BDF⁺02].

The decision problems of keys for XML are the subject of Chapter 5. We first show that any set of XML keys is always finitely satisfiable; that is, there exists a finite XML tree that satisfies the set. Then we turn our attention to the implication problem. Since key implication rely on path containment, we first present a sound and complete set of five inference rules for determining containment of our path language, and provide a quadratic time algorithm for testing inclusion of path expressions. We then provide a sound and

complete set of inference rules and a polynomial time algorithm for determining implication of three classes of key languages: weak absolute keys, weak absolute and relative keys considered together, and strong keys. The results on weak keys have been published in [BDF⁺03].

The propagation of XML keys to relations is considered in Chapter 6. First, a language for specifying transformations from XML to relations is presented. Despite its simplicity, it forms a core of many common transformations found in the literature. We then present two undecidability results that motivated our choices of transformation language and type of constraints considered: the first is that determining XML key propagation is undecidable when the transformation language can express all relational algebra operators; the second is that the propagation problem of keys and foreign keys is undecidable for any transformation language that can express identity mapping, which includes our language. Thus, we restrict our attention to the propagation of XML keys. We provide two algorithms: one is to check whether an FD is propagated from a set of XML keys via a predefined view, and the other is to compute a minimum cover of all FDs propagated from a set of XML keys. We then present our experimental results which show that these algorithms are efficient in practice. This Chapter is an extended version of [DFHQ03].

Chapter 7 concludes the dissertation by presenting a summary of our contributions and future work.

Chapter 2

Functional Dependencies for a Nested Relational Model

Data dependencies add semantics to a database schema and are useful for studying various problems such as database design, query optimization and how dependencies are carried into a view. In the context of the relational model, a wide variety of dependencies have been studied, such as functional, multivalued, join and inclusion dependencies (see [Mai83, AHV95] for overviews of this work). However, apart from notions of key constraints and inclusion dependencies [BFW98, PT99], dependencies in richer models than the relational model have not been as thoroughly studied. The functional dependency is one of the first and simplest form of dependencies identified. It states that if in a relation R two rows agree on the value of a set of attributes X then they must agree on the value of a set of attributes Y . In other words, in R , X determines Y . The dependency is written as $X \rightarrow Y$.

In the nested relational model, where the attributes of a relation may be sets rather than atomic types, we would like to be able to express dependencies that traverse into various levels of nesting. This motivated us to introduce a new form of functional dependencies, called *nested functional dependencies* (NFDs).

Nested functional dependencies can express a number of natural dependencies that arise

in a nested relational model. As an example of what we would like to be able to express, consider a type named *Course* defined as a set of records with attributes *cnum*, *time*, *students*, and *books*, where *students* is a set of records with labels *sid*, *age*, and *grade*, and *books* is a set of records with labels *isbn*, and *title*:

$$\begin{aligned} \textit{Course} : \{ & \langle \textit{cnum}, \textit{time}, \\ & \textit{students} : \{ \langle \textit{sid}, \textit{age}, \textit{grade} \rangle \}, \\ & \textit{books} : \{ \langle \textit{isbn}, \textit{title} \rangle \} \}. \end{aligned}$$

Some nested functional dependencies that we would like to be able to express for *Course* are:

1. *cnum* is a key.
2. Every *Course* instance is consistent on their assignment of *title* to a given *isbn*.
3. In a given course, each student gets a single grade.
4. The assignment of *age* to a given student must be consistent throughout the *Course* relation.
5. A student cannot be enrolled in courses that overlap on time.

Notice that there are “local” dependencies, as dependency 3, where a student can have only one *grade* for a given course, but a different *grade* for distinct courses. There are also “global” dependencies as dependencies 2 and 4, where the assignment of *title* to an *isbn*, and *age* to *sid* should be consistent throughout the *Course* relation. Dependency 5 illustrates how an attribute from an outer level of nesting is determined by attributes in a deeper level of nesting. Notice that even if every level of nesting presents a “key”, as suggested in [AB86], this type of dependency is not captured by the structure of the data.

As another example, consider the following example of a *Country* database, where a *country* can be composed either by states, as the U.S.A, or by a set of cities, as European countries.

$$\begin{aligned}
Country : \{ < name : \{string\}, \\
& continent : \{string\}, \\
& capital : \{string\}, \\
& states : \{ < sname : \{string\}, \\
& \quad scities : \{ < cname : \{string\}, \\
& \quad \quad location : \{ < latitude : \{int\}, \\
& \quad \quad \quad longitude : \{int\} > > > \}, \\
& \quad cities : \{ < cname : \{string\}, \\
& \quad \quad population : \{int\} > > \}
\}
\end{aligned}$$

Although this might look like a very badly designed database, it might be the approach used to model “sparse” data. In applications such as biological databases, due to the fact that the database is sparsely populated and evolves over time, most of the attributes are optional. By defining every attribute as a set, we can use empty sets as the value when the attribute is absent or undefined. The data model AceDB [TMD92], which is very popular among biologists, was developed with these properties in mind. Notice also that this example uses the fact that sets can be empty to express that a country can be either a set of states or a set of cities. Although this may not be the ideal way to express optional attributes, in cases where the data model does not support variants (as in this nested relational model), modeling optional attributes using empty sets is a feasible alternative.

Some dependencies that we would like to express in this example are:

- If a *country* is a set of *states*, then the country’s *name*, the state’s *sname* and the city’s *cname* determine the city’s *latitude* and *longitude*.
- If a *country* is a set of cities, then the country’s *name*, and the city’s *cname* determine the city’s *population*.

The main idea behind all these dependencies is that they follow paths. Using paths instead of attributes is a natural extension of the definition of functional dependencies in the relational model to a model where arbitrary levels of nesting are allowed.

This chapter is organized as follows. Section 2.1 formally defines nested functional dependencies. First, we introduce the nested relational model in Section 2.1.1, and our notion of path expressions in Section 2.1.2. We then define nested functional dependencies (NFDs) in Section 2.1.3 and present a translation of NFDs to logic in Section 2.1.4.

2.1 Functional Dependencies

The natural extension of a functional dependency $X \longrightarrow A$ for the nested relational model is to allow path expressions in X and A instead of attributes. That is, X is a set of paths and A is a single path. As an example, the requirement that a student's *age* in *Course* be consistent throughout the database could be written as $Course : [students : sid \rightarrow students : age]$, where “:” indicates traversal inside a set. Note that we have enclosed the dependency in square brackets “[]” and appended the name of the nested relation, *Course*.

We need to do two things prior to formally defining nested functional dependencies: The first is to give a definition of the nested relational model; The second is to describe a path language used to denote attributes in arbitrary levels of nesting.

2.1.1 Data Model

The nested relational model has been well studied (see [AHV95] for an overview). It extends the relational model by allowing the type of an attribute to be a set of records or a base type, rather than requiring it to be a base type (First Normal Form). For simplicity, we use the strict definition of the nested model and require that set and tuple constructors alternate, i.e. there are no sets of sets or tuples with a tuple component, although allowing nested records or sets does not substantially change the results established. For ease of presentation, we also assume that there are no repeated labels in a type, i.e., $\langle A : int, B : \{ \langle A : int \rangle \} \rangle$ is not allowed.

An example of a nested relation was given by *Course* in the previous section.

More formally, a nested relational database \mathcal{R} is a finite set of relation names, ranged over

by R_1, R_2, \dots . \mathcal{A} is a countable set of labels, ranged over by A_1, A_2, \dots , \mathcal{B} is a fixed finite set of base types, ranged over by \underline{b}, \dots

We introduce the data types **Types** as follows:

$$\tau ::= \underline{b} \mid \{\tau\} \mid \langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle$$

Here, \underline{b} are base types, like integers, and strings. The notation $\{\omega\}$ represents a set with elements of type ω , where ω must be a record type. $\langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle$ represents a record type with fields A_1, \dots, A_n of types τ_1, \dots, τ_n , respectively. Each τ_i must either be a base or a set type.

Definition 2.1 A database schema is a pair $(\mathcal{R}, \mathcal{S})$, where

- \mathcal{R} is a finite set of relation names
- \mathcal{S} is a schema mapping $\mathcal{S} : \mathcal{R} \rightarrow \mathbf{Types}$, such that for any $R \in \mathcal{R}$, $R \xrightarrow{\mathcal{S}} \tau^R$ where τ^R is a set of records in its outermost level.

Denotations of types. Let us denote by $\mathbf{D}^{\underline{b}}$ the domain of the base type \underline{b} , for any \underline{b} . The domain of our model \mathbf{D} is defined as the least set satisfying the equation:

$$\mathbf{D} \equiv \bigcup_{\underline{b}} \mathbf{D}^{\underline{b}} \cup \mathcal{A} \xrightarrow{\sim} \mathbf{D} \cup P_{fin}(\mathbf{D})$$

where $A \xrightarrow{\sim} B$ denotes the set of partial functions from A to B , and $P_{fin}(A)$ denotes the finite powerset of A .

Given a schema $(\mathcal{R}, \mathcal{S})$, the interpretation of each type τ in **Types**, $\llbracket \tau \rrbracket$, is defined by

$$\begin{aligned} \llbracket \underline{b} \rrbracket &\equiv \mathbf{D}^{\underline{b}} \\ \llbracket \{\tau\} \rrbracket &\equiv P_{fin}(\llbracket \tau \rrbracket) \\ \llbracket \langle A_1 : \tau_1, \dots, A_n : \tau_n \rangle \rrbracket &\equiv \{f \in \mathcal{A} \xrightarrow{\sim} \mathbf{D} \mid \text{dom}(f) = \{A_1, \dots, A_n\} \\ &\quad \text{and } f(A_i) \in \llbracket \tau_i \rrbracket, i = 1, \dots, n\} \end{aligned}$$

Definition 2.2 A database instance of a database schema $(\mathcal{R}, \mathcal{S})$ is a record I with labels in \mathcal{R} such that $\pi_R I$ is in $\llbracket \mathcal{S}(R) \rrbracket$ for each $R \in \mathcal{R}$.

We denote by $\mathcal{I}_{\mathcal{S}_c}$ the set of all instances of schema \mathcal{S}_c .

As an example, let $(\{Course\}, \mathcal{S})$ be a schema where

$$\mathcal{S}(Course) = \{ \langle cnum : string, time : int, students : \{ \langle sid : int, grade : string \rangle \} \rangle \}.$$

Then the following is an example of an instance of this schema:

$\langle Course \mapsto$

$$\begin{aligned} & \{ \langle cnum \mapsto "cis550", time \mapsto 10, students \mapsto \{ \langle sid \mapsto 1001, grade \mapsto "A" \rangle, \\ & \hspace{15em} \langle sid \mapsto 2002, grade \mapsto "B" \rangle \} \rangle, \\ & \langle cnum \mapsto "cis500", time \mapsto 12, students \mapsto \{ \langle sid \mapsto 1001, grade \mapsto "A" \rangle \} \rangle \} \end{aligned}$$

2.1.2 Path Expressions

We start by giving a very general definition of path expressions, and narrow them to be well-defined by a given type.

Definition 2.3 Let $\mathcal{A} = A_1, A_2, \dots$ be a set of labels. A path expression is a string over the alphabet $\mathcal{A} \cup \{:\}$. ϵ denotes the empty path.

Definition 2.4 A path expression p is well-typed with respect to type τ if

- $p = \epsilon$, or
- $p = Ap'$ and τ is a record type $\langle A : \tau', \dots \rangle$ and p' is well-typed with respect to τ' , or
- $p = :p'$ and τ is a set type $\{\tau'\}$ and p' is well-typed with respect to τ' .

As an example, $A : B$ is well-typed with respect to the type $\langle A : \{ \langle B : int, C : int \rangle \} \rangle$, but not with respect to the type $\langle A : int \rangle$.

Given an object e , the semantics of path expressions is given by:

$$\begin{aligned}
\llbracket \epsilon \ e \rrbracket &\equiv \llbracket e \rrbracket \\
\llbracket A \ e \rrbracket &\equiv \llbracket e \rrbracket(A) \\
\llbracket : \ e \rrbracket &\equiv \begin{cases} \text{undefined,} & \text{if } \llbracket e \rrbracket = \{\} \\ \llbracket e_1 \rrbracket, & \text{otherwise, where } \llbracket e_1 \rrbracket \text{ is an element of } \llbracket e \rrbracket \end{cases}
\end{aligned}$$

Note that the value of a path expression that traverses into an empty set is undefined, i.e., it does not yield a value in the database domain. We say that a path expression p is *well defined* on v if it always yields a value in the database domain.

As an example, if

$$v = \langle A \mapsto \{ \langle B \mapsto 10, C \mapsto 20 \rangle, \langle B \mapsto 15, C \mapsto 21 \rangle \} \rangle$$

then

- $A(v) = \{ \langle B \mapsto 10, C \mapsto 20 \rangle, \langle B \mapsto 15, C \mapsto 21 \rangle \}$
- $A : B(v) = 10$ or $A : B(v) = 15$

Observe that we could have adopted an alternative set semantics for path expressions of the form $A : B(v)$ in the previous example. That is, we could have defined the path expression to yield the set of all B attribute values of a set-valued attribute A – $A : B(v) = \{10, 15\}$. Consider now a second value $v' = \langle A \mapsto \{ \langle B \mapsto 10, C \mapsto 30 \rangle \} \rangle$. Using this set semantics, an equality test of the form $A : B(v) = A : B(v')$ would be interpreted as $A : B(v) \cap A : B(v') \neq \emptyset$, while in our interpretation the equality test returns true when $A : B(v) = 10$, and false otherwise. The motivation for adopting this interpretation is to be able to identify the $\langle B, C \rangle$ tuple in which the values of the B attribute coincide, which is important for defining the semantics for NFDs.

To help define nested functional dependencies, we introduce the notions of path prefix and size of a path expression.

Definition 2.5 Path expression p_1 is a prefix of p_2 if $p_2 = p_1 p'_2$. Path p_1 is a proper prefix of p_2 if p_1 is a prefix of p_2 and $p_1 \neq p_2$.

Definition 2.6 The size of a path expression of the form $p = A_1 : \dots : A_k$, denoted as $|p|$, is k , the number of labels in p .

2.1.3 Nested Functional Dependencies

With the notion of path expressions, we are now in a position to define nested functional dependencies (NFDs), and how an instance is said to satisfy an NFD.

Definition 2.7 Let $Sc = (\mathcal{R}, \mathcal{S})$ be a schema. A nested functional dependency (NFD) over Sc is an expression of the form $x_0 : [x_1, \dots, x_{m-1} \rightarrow x_m]$, $m \geq 1$, such that all x_i , $0 \leq i \leq m$, are path expressions of the form $A_1^i : \dots : A_{k_i}^i$, $k_i \geq 1$, where $x_0 = Ry$, $R \in \mathcal{R}$, and $y : x_i, 1 \leq i \leq m$, are well-typed path expressions with respect to τ^R .

In general, the base path x_0 can be an arbitrary path rather than just a relation name. For the degenerate case where $m = 1$, i.e. the NFD is of form $x_0 : [\emptyset \rightarrow x_m]$, then in any value of x_0 , x_m must be a constant.

Definition 2.8 Let $f = x_0 : [x_1, \dots, x_{m-1} \rightarrow x_m]$ be an NFD over schema Sc , I an instance of Sc , and v_1, v_2 two values of $x_0 : (I)$ in the database domain. I satisfies f , denoted $I \models f$, if for all v_1, v_2 , whenever

1. $x_i(v_1) = x_i(v_2)$ for all $i, 1 \leq i < m$
2. for every path x which is a common prefix of $x_i, x_j, 1 \leq i, j \leq m$, $x(v_1)$ coincide in $x_i(v_1)$ and $x_j(v_1)$ and $x(v_2)$ coincide in $x_i(v_2)$ and $x_j(v_2)$ (i.e. x_i and x_j follow the same path up to x in v_1 and in v_2)

then

$$x_m(v_1) = x_m(v_2)$$

If for some x_i , $1 \leq i \leq m$, $x_i(v_1)$, or $x_i(v_2)$ is undefined, we say f is trivially true.

In the next section, we give a translation of NFD to logic to precisely define its semantics.

This definition of NFDs is very broad, and captures many natural constraints. As an example, we can precisely state the constraints on *Course* described in the introduction of the chapter.

Example 2.1 In *Course*, *cnum* is a key.

Course : [*cnum* \rightarrow *time*]

Course : [*cnum* \rightarrow *students*]

Course : [*cnum* \rightarrow *books*] □

Example 2.2 For any two instances in *Course*, if they agree on *isbn* for some element of *books* then they must also agree on *title* for that element of *books*.

Course : [*books* : *isbn* \rightarrow *books* : *title*] □

Example 2.3 In a given course, each student gets a single grade.

Course : *students* : [*sid* \rightarrow *grade*] □

Note that in this example, *sid* is a “local” key to *grade*; this illustrates the use of a path rather than just a relation name outside the “[]”. Contrast this to the previous example, where the NFD requires that *isbn* and *title* should be consistent throughout the database.

Example 2.4 Every *Course* instance is consistent on their assignment of *age* to *sid*.

Course : [*students* : *sid* \rightarrow *students* : *age*] □

Example 2.5 A student cannot be enrolled in courses that overlap on time.

Course : [*time*, *students* : *sid* \rightarrow *cnum*] □

Some interesting properties of sets can also be expressed by NFDs. For example, if an instance I satisfies an NFD of the form $x_0 : [x_1 : x_2 \rightarrow x_1]$, then given two values v_1, v_2 of $x_0 : x_1(I)$, either $v_1 = v_2$, or $v_1 \cap v_2 = \emptyset$ ¹.

¹Note that values of $x_0 : x_1(I)$ must be of set type.

As an example, suppose that a university's courses database is defined as $Courses : \{\langle school, scourses : \{\langle cnum, time \rangle\} \rangle\}$, and it satisfies the NFD $Courses : [scourses : cnum \rightarrow school]$. We can conclude that *schools* in the university do not share course numbers, because the existence of the same *cnum* in different *schools* would violate the NFD.

NFDs can also express the fact that if a set is not empty then it must be a singleton. I.e., if an instance I satisfies an NFD of the form $x_0 : [x_1, \dots, x_m \rightarrow x_n : A]$, where x_n is not a proper prefix of any x_i , $1 \leq i \leq m$, then for any value v of $x_0 : (I)$ in which paths $x_1 \dots x_m$ are well-defined, all elements e of $x_n(v)$ have the same value for $A(e)$.

For example, let R be a relation with schema $\{\langle A : \{\langle B : int, C : int \rangle\}, D : int \rangle\}$. If $R : [D \rightarrow A : B]$, and $R : [D \rightarrow A : C]$, then it must be the case that A is either empty, or a singleton set, since for every value of A all elements agree on the values of B and C . Since these are the only attributes in A , then A has a single element.

It should be noted that our definition also allows some *unintuitive* NFDs. For example, assume $R : \{\langle A : int, B : \{\langle C : int, D : int \rangle\}, E : \{\langle F : int, G : int \rangle\} \rangle\}$. Then the NFD $R : [B : C \rightarrow E : F]$ implies that:

- all tuples $\langle F, G \rangle$ in E have the same value for F when B is not empty, and
- if any tuple $\langle C, D \rangle$ in B agrees on the value of C , then the elements $\langle F, G \rangle$ in E must have the same value for F .

Figure 2.1 shows an instance of R that does not satisfy $R : [B : C \rightarrow E : F]$. If we only consider the first tuple in the relation, the NFD is satisfied since all values of attribute F coincide, i.e. $B : C = 1$ determines $E : F = 5$. The existence of more than one value for F automatically invalidates the constraint because a single value in C would be related to distinct values in F as in the second tuple. This tuple also violates the dependency because it has a value in $B : C$ that also appears in the first tuple, but has a different value for $E : F$.

A	B		E	
	C	D	F	G
1	1	3	5	6
			5	7
2	2	2	3	4
	1	3	4	4

Figure 2.1: An instance that violates $R[B : C \rightarrow E : F]$.

2.1.4 NFDs expressed in logic

In the relational model, a functional dependency $Course : [cnum \rightarrow time, students]$ can be understood as the following formula:

$$\forall c_1 \in Course \forall c_2 \in Course \\ (c_1.cnum = c_2.cnum) \rightarrow (c_1.time = c_2.time \wedge c_1.students = c_2.students)$$

There is also a precise translation of NFDs to logic. Intuitively, given an NFD $R : [x_1 \dots x_{m-1} \rightarrow x_m]$, we introduce two universally quantified variables for R and for each set-valued attribute in $x_1 \dots x_m$ ². The body of the formula is an implication where the antecedent is the conjunction of equalities of the last attributes in $x_1 \dots x_{m-1}$ and the consequence is an equality of the last attribute in x_m .

As an example, $Course : [students : sid \rightarrow students : age]$ can be translated to the following formula:

$$\forall c_1 \in Course \forall c_2 \in Course \forall s_1 \in c_1.students \forall s_2 \in c_2.students. \\ (s_1.sid = s_2.sid \rightarrow s_1.age = s_2.age)$$

To formalize this translation, we define functions var and $parent$. Let $Sc = (\mathcal{R}, \mathcal{S})$ be a schema, and $f = x_0 : [x_1, \dots, x_{m-1} \rightarrow x_m]$ be an NFD defined over Sc , where $x_i = A_1^i :$

²It is a little more complicated for the general case where the base path can be an arbitrary path rather than R .

$\dots : A_{k_i}^i, 0 \leq i \leq m$, and $A_1^0 = R, R \in \mathcal{R}$.

Define *var* as a function that maps labels to variable names as follows:

- for each label A in τ^R that appears in some path $x_i, 0 \leq i \leq n$, $\text{var}(A) = v_A$. Recall that we assume labels cannot be repeated.

The function *parent* maps a label to the variable defined for its parent as follows:

- for all $A_1^i, 1 \leq i \leq m$, $\text{parent}(A_1^i) = \text{var}(A_{k_0}^0)$, i.e., the parent of the first labels in paths $x_1 \dots x_m$ is the variable associated with the last label in path x_0 .
- $\text{parent}(A_{j+1}^i) = \text{var}(A_j^i)$. Let $\{A_1^* \dots A_q^*\}$ be the set of such A_j labels, i.e., the set of labels that have some descendent in a path expression.

Also, let $\text{parent}(A_1^0).A_1^0 = R$. Then f is equivalent to the following logic formula:

$$\begin{aligned}
& \forall v_{A_1^0} \in \text{parent}(A_1^0).A_1^0 \dots \forall v_{A_{k_0-1}^0} \in \text{parent}(A_{k_0-1}^0).A_{k_0-1}^0 \\
& \forall v_{A_{k_0}^0} \in \text{parent}(A_{k_0}^0).A_{k_0}^0 \quad \forall v_{A_{k_0}^0}^2 \in \text{parent}(A_{k_0}^0).A_{k_0}^0 \\
& \forall v_{A_1^*}^1 \in \text{parent}(A_1^*)^1.A_1^* \quad \forall v_{A_1^*}^2 \in \text{parent}(A_1^*)^2.A_1^* \dots \\
& \forall v_{A_q^*}^1 \in \text{parent}(A_q^*)^1.A_q^* \quad \forall v_{A_q^*}^2 \in \text{parent}(A_q^*)^2.A_q^* \\
& ((\text{true} \wedge \\
& \quad \text{parent}(A_{k_1}^1)^1.A_{k_1}^1 = \text{parent}(A_{k_1}^1)^2.A_{k_1}^1 \wedge \dots \wedge \\
& \quad \text{parent}(A_{k_{m-1}}^{m-1})^1.A_{k_{m-1}}^{m-1} = \text{parent}(A_{k_{m-1}}^{m-1})^2.A_{k_{m-1}}^{m-1}) \\
& \rightarrow \\
& (\text{parent}(A_{k_m}^m)^1.A_{k_m}^m = \text{parent}(A_{k_m}^m)^2.A_{k_m}^m)
\end{aligned}$$

Note that only one variable is mapped to each label in $A_1^0, \dots, A_{k_0-1}^0$, whereas two variables are used elsewhere.

Using this translation, Examples 2.1.3 and 2.1.3 can be expressed as:

- *Course* : [*books* : isbn \rightarrow *books* : title]
- $$\forall c_1 \in \text{Course} \quad \forall c_2 \in \text{Course} \quad \forall b_1 \in c_1.\text{books} \quad \forall b_2 \in c_2.\text{books}.$$
- $$(b_1.\text{isbn} = b_2.\text{isbn} \rightarrow b_1.\text{title} = b_2.\text{title})$$

Note that *books* is referred to twice in the dependency, and two variables for *books* are introduced in the logical form.

- $Course : students : [sid \rightarrow grade]$
 $\forall c \in Course \ \forall s_1 \in c.students \ \forall s_2 \in c.students$
 $(s_1.sid = s_2.sid \rightarrow s_1.grade = s_2.grade)$

Note that only one variable is introduced for *Course*, and two variables are introduced for *students*, the last label in x_0 .

2.2 Discussion

In the definition of NFDs, the base path can be an arbitrary path rather than just a relation name. The motivation for allowing this is to syntactically differentiate between local and global functional dependencies: $R : A : [B \rightarrow C]$ is a *local* functional dependency in A , while $R : [A : B \rightarrow A : C]$ defines a *global* dependency between B and C . However, the local dependency is provably equivalent³ to the dependency $R : [A, A : B \rightarrow A : C]$. Intuitively, by requiring equality on A (as a set), the dependency between B and C becomes local to the set. Therefore, the expressive power of NFDs with arbitrary paths and relation names as base paths are the same. However, we believe that the first form is more intuitive.

Most of the early work on functional dependencies (FDs) adopted a more restrictive nested relational model than the one presented in Section 2.1.1. In [AB86] every level of nesting is required to have at least one atomic attribute, while in [OY87] and [MNE96] relations are required to be in *partition normal form*[RKS88]. That is, the atomic attributes in each level of nesting form a key. In these earlier work, the definition of FDs is either the one given for the relational model [OY87] and are defined on atomic attributes, or extends it by allowing equality on sets [Mak77]. Our definition clearly subsumes these definitions. Moreover, with a single notation, our definition of NFDs can express both the notions of functional dependencies and multi-valued dependencies on nested relations. That is, an NFD $R : [X \rightarrow y]$ corresponds to a functional dependency if the type of y is atomic, and

³The equivalence of these two forms is proved in Chapter 3.

to a multi-valued dependency otherwise. A detailed discussion on the correspondence of NFDs and functional dependencies and multi-valued dependencies is presented in the next Chapter.

The idea of extending functional dependencies to allow path expressions instead of simple attribute names have been investigated by Weddell [Wed92] and also by Tari et al. [TSS97] in the context of an object-oriented data model. While Weddell's work supports a data model of classes, where each class is associated with a simple type (a flat record type), our model supports a nested relational model with arbitrary levels of nesting. In [Wed92], following a path entails an implicit "dereference" operation, while in NFDs following a path means traversal into an element of a nested set. They present a set of inference rules and prove they are complete. We believe this work and ours are complementary and that it would be interesting to investigate how the two approaches could be combined into a single framework.

In [TSS97], more general forms of functional dependencies for the object-oriented model are proposed. Their model supports nested sets, and classes of objects, and the dependencies allow inter- and intra-set dependencies, and also dependencies between objects without specifying an specific path. For example, it is possible to express that any path between two objects should lead to the same value. But, as opposed to our model, they assume that every level of nesting presents a key or an object ID. Inference rules for the proposed forms of functional dependencies are presented, but they do not claim or prove their completeness.

We believe our definition of NFDs naturally extends the definition of functional dependencies for the relational model by using path expressions instead of attribute names. The meaning of NFDs was given by defining their translation to logic. NFDs allow the definition of both local and global functional dependencies (intra and inter-set dependencies). They can also express some properties of sets. For example, it is possible to express that sets do not share some values, and that a given set is a singleton, as exemplified in the previous section. The importance of singleton sets is evident in [FSTG85], which investigates when functional dependencies are maintained or destroyed when relations are nested and

unnested. In most cases, this relies on knowing whether a set is a singleton or multivalued.

More recently, a number of definitions of functional dependencies for XML has been proposed [CDHZ03, LVL03, VL03, AL04]. As we will show in more detail in Chapter 4, an XML document can be seen as a node-labeled tree with values on the leaves. These notions of FDs also involve paths, but as opposed to NFDs, FDs for XML involve not only value equality, but also node identity. A detailed discussion on FDs for XML will be presented in Chapter 4, after defining of our notion of keys for XML.

Chapter 3

Reasoning about Nested Functional Dependencies

One of the most interesting questions involving NFDs is that of logical implication, i.e., deciding if a new dependency holds given a set of existing dependencies. This problem can be addressed from two perspectives: One is to develop algorithms to decide logical implication, for example, tableau chase techniques (see [MMS79] for the relational model, and more recently [PT99, PDST00] for a complex object model, and [DT03b] for XML). The other is to develop inference rules that allow us to derive new dependencies from the given ones.

The development of inference rules is important for many reasons [BV84a]: First, it helps us gain insight into the dependencies. Second, it may help in discovering efficient decision procedures for the implication problem. Third, it provides tools to operate on dependencies. For example, in the relational model, it provided the basis for testing equivalence preserving transformations, such as lossless-join decomposition, and dependency preserving decomposition, which lead to the definition of normal forms of relations, a somewhat more mechanical way to produce a database design [Ull83].

$\frac{Y \subseteq X}{R : [X \rightarrow Y]}$	(reflexivity)
$\frac{R : [X \rightarrow Y]}{R : [XZ \rightarrow YZ]}$	(augmentation)
$\frac{R : [X \rightarrow Y] \quad R : [Y \rightarrow Z]}{R : [X \rightarrow Z]}$	(transitivity)

Table 3.1: Armstrong Axioms for FD implication

3.1 Axiomatization for NFD Implication

In the relational model, a simple set of three rules – called Armstrong’s Axioms – are sound and complete for functional dependencies (FDs). They are presented in Table 3.1, using our notation, where “paths” are single attributes, and X , Y , and Z denote sets of attributes.

The logical implication problem for these rules is formally defined as:

Definition 3.1 *Let Sc be a schema, Σ be a set of FDs over Sc , and σ an FD over Sc . Σ logically implies σ under Sc , denoted $\Sigma \models_{Sc} \sigma$ if for all instances I of Sc , $I \models \Sigma$ implies $I \models \sigma$.*

The implication problem for NFDs that we will consider is slightly changed from that for FDs: no instances are allowed to contain empty sets. Empty sets cause tremendous difficulties in reasoning since formulas such as

$$\forall x \in R. P(x)$$

are trivially true when R is empty. These problems are discussed in detail in Section 3.2. For completeness, we state below the implication problem that we are considering for NFDs.

Definition 3.2 *Let Sc be a schema, Σ be a set of NFDs over Sc , and σ an NFD over Sc . Σ logically implies σ under Sc , denoted $\Sigma \models_{Sc} \sigma$ if for all instances I of Sc **with no empty sets**, $I \models \Sigma$ implies $I \models \sigma$.*

$\frac{x \in X}{x_0 : [X \rightarrow x]}$	(reflexivity)
$\frac{x_0 : [X \rightarrow z]}{x_0 : [XY \rightarrow z]}$	(augmentation)
$\frac{x_0 : [X \rightarrow x_i], i \in [1, n], \quad x_0 : [x_1, \dots, x_n \rightarrow y]}{x_0 : [X \rightarrow y]}$	(transitivity)
$\frac{x_0 : y : [X \rightarrow z]}{x_0 : [y, y : X \rightarrow y : z]}$	(push-in)
$\frac{x_0 : [y, y : X \rightarrow y : z]}{x_0 : y : [X \rightarrow z]}$	(pull-out)
$\frac{x_0 : [A : X, B_1, \dots, B_k \rightarrow A : z]}{x_0 : A : [X \rightarrow z]}$	(locality)
$\frac{x_0 : [x_1 : A, x_2, \dots, x_k \rightarrow y], \quad x_1 \geq 1, \quad x_1 \text{ is not prefix of } y}{x_0 : [x_1, x_2, \dots, x_k \rightarrow y]}$	(prefix)
$\frac{x_0 : [x \rightarrow x : A_1], \dots, x_0 : [x \rightarrow x : A_n], \quad \text{type of } x \text{ is } \{<A_1, \dots, A_n>\}}{x_0 : [x : A_1, \dots, x : A_n \rightarrow x]}$	(singleton)

Table 3.2: Rules for NFD implication

In this section, we present a sound and complete set of inference rules for NFDs in the restricted case in which no empty sets are present in any instance. The extension to allow empty sets in instances is discussed in detail in Section 3.2. Conceptually, the rules can be broken up into three categories: The first three mirror Armstrong’s axioms – reflexivity, augmentation and transitivity. The next two transform between alternate forms of NFDs discussed at the end of the last chapter – push-in and pull-out.¹ The last three allow inferences based solely on the nested form of the data – locality, prefix, and singleton.

In the following, x, y, z, x_0, x_1, \dots are path expressions, and $A_1, A_2, \dots, B_1, B_2, \dots$ are attribute labels. XY denotes $X \cup Y$, where X, Y are sets of path expressions, and $x : X$ denotes the set $\{x : x_1, \dots, x : x_k\}$, where $X = \{x_1, \dots, x_k\}$.

The **NFD-rules** are presented in Table 3.2.

¹A discussion of why we don’t just adopt a simpler form for NFDs which would eliminate these two rules is deferred to section 3.3.

Example 3.1 Let R be a relation with schema $\{\langle A : \{\langle B : \{\langle C \rangle\}, E : \{\langle F, G \rangle\} \rangle \rangle, D \rangle\}$, on which the following NFDs are defined:

(nfd1) $R : [A : B : C, D \rightarrow A : E : F]$

(nfd2) $R : A : [B \rightarrow E : G]$

We claim that $R : A : [B \rightarrow E]$. The proof using the NFD-rules follows.

1. $R : A : [B : C \rightarrow E : F]$ by locality of nfd1.

The locality rule allows us to derive a local NFD from a global one, by dismissing the attributes outside the level of nesting of the local NFD. In the example above, notice that for any element in R , given a value of A there exists a unique value of D , since they are labels in a record type. Therefore, locally for any value of A , $B : C \rightarrow E : F$.

2. $R : A : [B \rightarrow E : F]$ by prefix rule on (1).

(1) states that whenever two tuples in R have a common value for C in the set B , then the value of $E : F$ must also agree. In particular, if two tuples agree on the value of B then they present a common element, since we assumed that there are no empty sets in instances of R .

3. $R : A : E : [\emptyset \rightarrow F]$ by locality of (2).

If in any tuple in $R : A$, the value of B determines the value of $E : F$ then every element in E have to agree on the value of F , otherwise (2) would be violated. Therefore, locally in any $A : E$ the value of F is constant.

4. $R : A : [E \rightarrow E : F]$ by push-in.

If the value of F is constant inside any value of $A : E$, for any given value of $A : E$ there exists a unique value of F . Therefore, the whole set determines the value of F .

5. $R : A : E : [\emptyset \rightarrow G]$ by locality of nfd2.

6. $R : A : [E \rightarrow E : G]$ by push-in.

7. $R : A : [E : F, E : G \rightarrow E]$ by singleton with (4) and (6).

Since the value of the set E determines the value of each of its attributes, then E must be a singleton. Therefore, the values of its unique element determines the value of the set.

8. $R : A : [B \rightarrow E]$ by transitivity with (7), (2), and nfd2. □

Lemma 3.1 *Let Sc be a schema. The NFD-rules are sound for logical implication of NFDs under Sc for the case when no empty sets are present in a instance.*

Proof.

1. **reflexivity:** Suppose $f \equiv x_0 : [X \rightarrow x]$ is not satisfied for some $x \in X$. Let v_1, v_2 be two arbitrary values of $x_0 : (I)$. If for some $y \in X$, $y(v_1) \neq y(v_2)$, then v_1, v_2 can not violate f . If for all $y \in X$ $y(v_1) = y(v_2)$, and $x(v_1) \neq x(v_2)$, v_1, v_2 violates f . But $x \in X$, therefore $x(v_1) = x(v_2)$.
2. **augmentation:** Suppose I satisfies $f_1 \equiv x_0 : [X \rightarrow z]$, but not $f_2 \equiv x_0 : [XY \rightarrow z]$. Let v_1, v_2 be two arbitrary values of $x_0 : (I)$. Suppose for all $y \in Y$, $y(v_1) = y(v_2)$, and for all $x \in X$, $x(v_1) = x(v_2)$, yet $z(v_1) \neq z(v_2)$. But since f_1 is satisfied and for all $x \in X$, $x(v_1) = x(v_2)$, $z(v_1) = z(v_2)$, a contradiction.
3. **transitivity:** Suppose I satisfies $f_1 \equiv x_0 : [X \rightarrow x_1], \dots, f_n \equiv x_0 : [X \rightarrow x_n], f_y \equiv x_0 : [x_1, \dots, x_n \rightarrow y]$. Yet, I does not satisfy $f \equiv x_0 : [X \rightarrow y]$. Let v_1, v_2 be two arbitrary values of $x_0 : (I)$. Suppose $p(v_1) = p(v_2)$ for all $p \in X$. Since I satisfies f_i $x_i(v_1) = x_i(v_2)$ for all x_i , $1 \leq i \leq n$. But I also satisfies f_y , therefore $y(v_1) = y(v_2)$, and I satisfies f .
4. **push-in:** Suppose I satisfies $f_1 \equiv x_0 : y : [X \rightarrow z]$, but not $f_2 \equiv x_0 : [y, y : X \rightarrow y : z]$. Let v_1, v_2 be two arbitrary values of $x_0 : (I)$, such that $y(v_1) = y(v_2)$, and e_1, e_2 be elements in $y(v_1)$, and in $y(v_2)$, respectively, such that for all $x \in X$, $x(e_1) = x(e_2)$, and $z(e_1) \neq z(e_2)$. But $\{e_1, e_2\} \subseteq y(v_1) = y(v_2)$, and since I satisfies f_1 , $z(e_1) = z(e_2)$.

5. **pull-out:** Suppose I satisfies $f_1 \equiv x_0 : [y, y : X \rightarrow y : z]$, but not $f_2 \equiv x_0 : y : [X \rightarrow z]$. Let v_1 be an arbitrary value of $x_0 : (I)$, and e_1, e_2 two elements in $y(v_1)$ such that for all $x \in X$, $x(e_1) = x(e_2)$, and $z(e_1) \neq z(e_2)$. But from f_1 if $y(v_1) = y(v_1)$ and $x(e_1) = x(e_2)$ for all $x \in X$ then $z(e_1) = z(e_2)$, a contradiction.
6. **locality:** Suppose I satisfies $f_1 \equiv x_0 : [A : x_1, \dots, A : x_m, B_1, \dots, B_k \rightarrow A : x_n]$, but not $f_2 \equiv x_0 : A : [x_1, \dots, x_m \rightarrow x_n]$. Let r be an arbitrary value of $x_0 : (I)$, and v_1, v_2 arbitrary values of $A : (r)$. Suppose $x_i(v_1) = x_i(v_2)$ for all x_i , $1 \leq i \leq m$, yet $x_n(v_1) \neq x_n(v_2)$. But $x_i(v_1), x_i(v_2)$ are values of $A : x_i(r)$, and since r is a record with labels A, B_1, \dots, B_k there is only one value for all B_i , $1 \leq i \leq k$. Since I satisfies f_1 , $x_n(v_1) = x_n(v_2)$.
7. **prefix:** Suppose I satisfies $f_1 \equiv x_0 : [x_1 : A, x_2, \dots, x_k \rightarrow y]$, $|x_1| \geq 1$, but I does not satisfy $f_2 \equiv x_0 : [x_1, x_2, \dots, x_k \rightarrow y]$. Let v_1, v_2 be two arbitrary value of $x_0 : (I)$. Suppose for all x_i , $1 \leq i \leq k$, $x_i(v_1) = x_i(v_2)$, but $y(v_1) \neq y(v_2)$. Since $x_1(v_1) = x_1(v_2)$, for every element $e_1 \in x_1(v_1)$ there exists an element $e_2 \in x_1(v_2)$ such that $x_1 : A(v_1) = x_1 : A(v_2)$. The value of $y(v_1), y(v_2)$ does not depend on the elements e_1, e_2 chosen because x_1 is not prefix of y by assumption. Given that I satisfies f_1 , $y(v_1) = y(v_2)$, which contradicts our initial assumption. Hence, $x_0 : [x_1 \dots x_k \rightarrow y]$.
8. **singleton:** Note first that if the value of a set x is proven to be a singleton, then the unique element of the set determines the value of the set. In particular, if the element of the set is a record then the set of attributes of the record, $\{A_1, \dots, A_n\}$, determines the value of the set, i.e., $x_0 : [x : A_1, \dots, x : A_n \rightarrow x]$. Suppose I satisfies $f_1 \equiv x_0 : [x \rightarrow x : A_1], \dots, f_n \equiv x_0 : [x \rightarrow x : A_n]$. Yet, I does not satisfy $f \equiv x_0 : [x : A_1, \dots, x : A_n \rightarrow x]$. We'll show that under the assumptions x is a singleton. Suppose not. Let v_1 be an arbitrary value of $x_0 : x(I)$, and let e_1, e_2 be two elements in v_1 . There must exist some A_i such that $A_i(e_1) \neq A_i(e_2)$. But for all i , $1 \leq i \leq n$, I satisfies $x_0 : [x \rightarrow x : A_i]$, and therefore $A_i(e_1) = A_i(e_2)$. As a consequence $e_1 = e_2$, and x is a singleton. \square

There are several rules that are consequences of the rules defined above. Here we give just

one that will be useful in later discussions.

• **full-locality:**

if

1. $x_0 : [x : X, Y \rightarrow x : z]$
2. x is not a proper prefix of any $y \in Y$

then $x_0 : [x, x : X \rightarrow x : z]$.

Proof. Let $x = A_1 : \dots : A_k$, i.e., we can rewrite the NFD as $x_0 : [Y, A_1 : \dots : A_k : X \rightarrow A_1 :, wts : A_k : z]$. Applying the prefix rule multiple times on paths in Y . We get $x_0 : [B_1, \dots, B_m, A_1 : Y_1, \dots, A_1 : \dots : A_{k-1} : Y_{k-1}, A_1 : \dots : A_k : X \rightarrow A_1 : \dots : A_k : z]$, where for all $y \in Y_i$, $1 \leq i < k$, $|y| = 1$, and for all $p \in \{B_1, \dots, B_m\} \cup A_1 : Y_1 \cup \dots \cup A_1 : \dots : A_{k-1} : Y_{k-1}$ there exists a $q \in Y$ such that $q = pq'$. We can then apply the locality rule and get $x_0 : A_1 : [Y_1, \dots, A_2 : \dots : A_{k-1} : Y_{k-1}, A_2 : \dots : A_k : X \rightarrow A_2 : \dots : A_k : z]$. Applying locality rule $k - 1$ more times we get $x_0 : A_1 : \dots : A_k : [X \rightarrow z]$. Then by push-in we finally obtain $x_0 : [x, x : X \rightarrow x : z]$ \square

3.1.1 Completeness of the NFD-rules

In order to prove completeness, we need to define the set of paths in a schema, and the closure of a set of paths.

Definition 3.3 *Let $Sc = (\mathcal{R}, \mathcal{S})$ be a schema. Then the paths of Sc , denoted as $Paths(Sc)$, is the set of all path expressions $p \equiv Rp'$, such that $R \in \mathcal{R}$, and p' is well-typed with respect to τ^R . Similarly, the paths of R , $R \in \mathcal{R}$, denoted as $Paths_{Sc}(R)$, is the set of paths p such that $p \in Paths(Sc)$, and $p \equiv Rp'$.*

Definition 3.4 *Let Sc be a schema, Σ a set of NFDs over Sc , x_0 a path expression, and $X = \{x_1, \dots, x_n\}$ a set of path expressions such that $\{x_0, x_0 : x_1, \dots, x_0 : x_n\} \subseteq Paths(Sc)$. The closure of X under x_0 , and Σ , denoted $(x_0, X, \Sigma)^{*, Sc}$ (or $(x_0, X)^*$ when Σ and Sc are*

understood) is the set of paths $x_0 : q$ such that $x_0 : q \in Paths(Sc)$, and $x_0 : [X \rightarrow q]$ can be derived from the NFD-rules.

Let $Sc = (\mathcal{R}, \mathcal{S})$ be a schema, Σ a set of NFDs over Sc , and $X \cup \{x_0, x\}$ a set of paths in $Paths(R)$, where $R \in \mathcal{R}$. The completeness proof is based on the construction of an instance I of R such that $I \models \Sigma$, but $I \not\models x_0 : [X \rightarrow x]$ if $x \notin (x_0, X, \Sigma)^{*, Sc}$. In the following we describe the construction of I .

The main idea in building I is to create a pair of tuples in each level of the hierarchy that agree on the attributes that are in the closure $(x_0, X, \Sigma)^{*, Sc}$, and disagree on all others. When the type of the attribute is either a base type, or a set of a base type, it is sufficient to pick values or singleton sets containing values that coincide or not, depending on its membership in the closure. But for attributes that are of type set of tuples, it is not so simple. Our strategy for this problem is to pre-compute the value of all attributes in the closure and use them to populate the instance I whenever their values must coincide. The pre-computed value for an attribute of a base type is an arbitrary value, and for an attribute that is of type set of a base type, it is a singleton set containing an arbitrary value. When the type of the attribute is a set of records, the computed value contain tuples that disagree on the value of attributes that are not in the closure. More specifically, let A be an attribute of type $\{ \langle A_1, \dots, A_k \rangle \}$, where A is in the closure. The pre-computed value of A will be a singleton set if for all A_i , $1 \leq i \leq k$, A_i is in the closure. If there exists an A_i not in the closure, A will be composed of two tuples that coincide on the values of the A_i 's in the closure, and disagree on all others.

Given that all attributes in the closure have a pre-computed value, the initial idea of creating a pair of tuples for each level of the hierarchy can be applied: whenever an attribute is in the closure, it is populated with its pre-computed value; otherwise, a fresh new value is created. Yet, there is a situation when the technique fails: when an attribute A of type set of records is not in the closure, but all the element attributes are part of the closure. In this case, if the strategy above is used to create two values for attribute A in different tuples, they would be value equal, since all its element attributes are in the closure. But since A is not in the closure they should not be equal. To overcome this problem, the

value of A contains an additional tuple with fresh new values. The only detail that must be observed is that some of the attributes may be required to be constants in A . This is true when for an attribute A_i of A the set of constraints Σ implies that $x_0 : \dots : A[\emptyset \rightarrow A_i]$. In this case, the pre-computed value should be assigned to A_i in the new tuple, so that it agrees with the value of A_i in the first tuple.

The algorithm for constructing I is shown in Figure 3.1. We assume that the domain of all base types are infinite, and to make the exposition simpler, we consider a unique base type b . The pre-computed values for paths p in the closure are stored in global variables named $value(p)$. Note that if p is of type set of records and in its construction $value(p')$ is used (this happens when p is prefix of p') then $value(p')$ should be thought as a placeholder until its value is evaluated. $newValue()$ is a function that returns a fresh new value in the domain of b , and $type_{Sc}(p)$ is a function that returns the type of the attribute reached by following p in a schema Sc . We say that $type_{Sc}(p)$ is the *type of path expression p under Sc* .

The algorithm starts by computing the pre-computed values for every path in the closure based on a single value val , and using function $assignVal$. If the path is of type set of records, function $assignVal$ returns a set of two tuples $\{r_1, r_2\}$ that coincide in the values of attributes in the closure. Note that if all element attributes of p are in the closure, $r_1 = r_2$, and therefore the function returns a singleton set. The construction of instance I starts by calling function $buildX_0$, which assigns single fresh values to every path that is not prefix of the base path x_0 , and then calls function $assignVal$ to create a pair of tuples for the base path x_0 . Creating fresh new values for paths that are not in the closure is the purpose of function $computeNew$. Observe that whenever a path p of type $\{<A_1, \dots, A_k>\}$ is not in the closure but all the A_i are part of the closure, the function $newRow$ is called to create a new tuple so that no two values of $x_0 : p(I)$ coincide.

To illustrate the algorithm described, consider the following examples.

Example 3.2 Let R be a relation with schema $\{< A, B : \{< C >\}, D, E : \{< F, G >\}, H : \{< J, L >\}, I, M : \{< N, O >\} >\}$. The set Σ of NFDs defined for R are:

$$R : [A \rightarrow B : C]$$

```

Algorithm buildInstance ( $\Sigma, x_0, X$ )
  Input:  $\Sigma$ , a set of NFDs;  $x_0$ , a base path;  $X$ , a set of paths.
  Output: an instance  $I$  such that  $I \models \Sigma$  and  $I \not\models x_0 : [X \rightarrow q]$  if  $x_0 : q \notin (x_0, X, \Sigma)^*$ 
   $closure := (x_0, X, \Sigma)^*$ ;      /* pick a new value and pre-compute values for paths in  $closure^*$ /
   $val := newValue()$ ;
  for all  $p \in closure$  do  $value(p) := assignVal(val, p)$ ;
  return ( $buildX_0(R)$ );          /* build a pair of tuples for  $x_0$  */

function buildX0 ( $p$ )
  Input:  $p$ , a path expression of type  $\{<A_1, \dots, A_n>\}$ .
  Output: an instance  $I$  up to the base path  $x_0$ .
  if  $p = x_0$  then return  $assignVal(0, x_0)$ ;
  for all  $A_i, 1 \leq i \leq n$ , do
    if  $p : A_i$  is prefix of  $x_0$  then  $r.A_i := buildX_0(p : A_i)$ ; else  $r.A_i := computeNew(p : A_i)$ ;
  return  $\{r\}$ ;

function assignVal ( $val, p$ )
  Input:  $val$ , a value to be assigned to  $p$ ;  $p$ , a path expression.
  Output: a value for  $p$ , according to its type.
  case  $type_{Sc}(p)$  of
   $b$ :          return  $val$ ;
   $\{b\}$ :      return  $\{val\}$ ;
   $\{<A_1, \dots, A_n>\}$ : for all  $A_i, 1 \leq i \leq n$ , do
    if  $p : A_i \in closure$ 
    then  $r_1.A_i := value(p : A_i)$ ;  $r_2.A_i := value(p : A_i)$ ;
    else  $r_1.A_i := computeNew(p : A_i)$ ;  $r_2.A_i := computeNew(p : A_i)$ ;
  return  $\{r_1, r_2\}$ ;

function computeNew ( $p$ )
  Input:  $p$ , a path expression.
  Output: a fresh new value for  $p$  according to its type.
  case  $type_{Sc}(p)$  of
   $b$ :          return  $newValue()$ ;
   $\{b\}$ :      return  $\{newValue()\}$ ;
   $\{<A_1, \dots, A_n>\}$ : for all  $A_i, 1 \leq i \leq n$ , do
    if  $p : A_i \in closure$  then  $r.A_i := value(p : A_i)$ ;
    else  $r.A_i := computeNew(p : A_i)$ ;
    if  $\{p : A_1, \dots, p : A_n\} \subseteq closure$  then return  $\{r, newRow(p, (p, \emptyset)^*)\}$ ;
    else return  $\{r\}$ ;

function newRow ( $p, constantAtt$ )
  Input:  $p$ , path expression of type  $\{<A_1, \dots, A_n>\}$ ;
     $constantAtt$ , a set of paths that have constant values in  $p$ .
  Output: a tuple  $r$  with pre-computed values for attributes in  $constantAtt$ .
  for all  $A_i, 1 \leq i \leq n$ , do
    if  $p : A_i \in constantAtt$  then  $r.A_i := value(p : A_i)$ ;
    else case  $type_{Sc}(p : A_i)$  of
       $b$ :           $r.A_i := newValue()$ ;
       $\{b\}$ :       $r.A_i := \{newValue()\}$ ;
       $\{<B_1, \dots, B_k>\}$ :  $r.A_i := \{newRow(p : A_i, constantAtt)\}$ ;
  return  $r$ ;

```

Figure 3.1: Algorithm for building an instance I of $Sc(R)$

$$\begin{aligned}
R &: [B : C \rightarrow D] \\
R &: [D \rightarrow E : F] \\
R &: [A \rightarrow E : G] \\
R &: [B : C \rightarrow H] \\
R &: [I \rightarrow H : J]
\end{aligned}$$

Then, using the NFD-rules, the computed closure $(R, \{B\}, \Sigma)^*$ is the set $\{R : B, R : B : C, R : D, R : E : F, R : H, R : H : J\}$, and the following instance is constructed using algorithm *buildInstance*, where attributes in the closure are shown in boldface.

A	B	D	E		H		I	M	
3	C 0	0	F 0	G 5	J 0	L 1	{7}	N 9	O 10
4	C 0	0	F 0	G 6	J 0	L 1	{8}	N 11	O 12

Observe that in this example, every value of $R : B : C(I)$ is a singleton set, since both $R : B$, and $R : B : C$ are in the closure; on the other hand, $R : H(I)$ contains two tuples because there exists an attribute element $R : H : L$ that is not in the closure. \square

Example 3.3 Let R be a relation with schema $\langle A : \langle B : \langle C, D, E : \langle F, G \rangle \rangle \rangle, H \rangle$. The set Σ of NFDs defined for R are:

$$\begin{aligned}
R &: [A : B : C \rightarrow A : B] \\
R &: [A : B : C \rightarrow A : B : E : F] \\
R &: [H \rightarrow A : B : D]
\end{aligned}$$

Then, $(R, \{A : B : C\}, \Sigma)^* = \{R : A : B : C, R : A : B, R : A : B : D, R : A : B : E : F\}$.

The following instance is constructed using the algorithm presented.

A			H
B			
C	D	E	
		F	G
0	0	0	1
		F	G
0	0	0	2
B			
C	D	E	
		F	G
3	0	5	6
B			
C	D	E	
		F	G
0	0	0	1
		F	G
0	0	0	2
B			
C	D	E	
		F	G
7	0	9	10

This example illustrates the construction of I when a set-valued attribute A is not in the the closure, but all its elements, in this case only $A : B$ is part of the closure. Each value of $A(I)$ contains two values for element attribute B : one containing its pre-computed value, and the other containing a fresh set value. □

Given that the completeness proof is based on the construction of an instance I such that $I \models \Sigma$, we can say that for any finite set Σ of NFDs, one can always find a finite instance of the schema that satisfies Σ . This is referred to as the *finite satisfiability property*.

Observation. Any finite set of NFDs is finitely satisfiable.

In order to simplify the completeness proof, we first make a number of observations about the properties of an instance I constructed as a result of the algorithm in Figure 3.1. Observations that are direct consequences from the construction of I are stated without a proof.

Observation 3.1 *For any path p , if x_0 is not a prefix of p then there is a unique value of $p(I)$.*

Observation 3.2 *Let p be a path of type set of records. If p is in closure or $p = x_0$ then $p(I)$ is a singleton set if all its attributes are also in closure; otherwise, it contains two elements. If p is not in closure then $p(I)$ contains two elements if all its attributes are in closure; otherwise, it is a singleton set.*

Observation 3.3 *Let p be a path. If p is in closure then $p(I)$ is either built by function `assignVal` or `newRow`. If p is not in closure then $p(I)$ is either built by function `computeNew` or `newRow`.*

Observation 3.4 *If $pq \notin (p, \emptyset)^*$ then values resulting from `newRow(pq, (p, \emptyset)^*)` are distinct from each other. As a consequence, values returned by `computeNew(p)` are also distinct.*

Proof. By induction on the structure of pq .

Base Case: If type of pq is a base type or a set of base types and $pq \notin (p, \emptyset)^*$, then the value returned by `newRow` is given by `newValue()`, which is a value distinct from any other in I .

Inductive Step: Let type of $p : q$ be $\{<A_1, \dots, A_n>\}$. Suppose, by contradiction, that the resulting value is not unique. This can only happen if the value assigned to all element attributes A_i is `value(p : q : A_i)`. That is, for all A_i , $1 \leq i \leq n$, $p : q : A_i \in (p, \emptyset)^*$. Let $p \equiv x_0 : p'$. By the full-locality rule, $x_0 : [p' : q \rightarrow p' : q : A_i]$ for all A_i , $1 \leq i \leq n$. Then by singleton, $x_0 : [p' : q : A_1, \dots, p' : q : A_n \rightarrow p' : q]$. By transitivity, $x_0 : [p' \rightarrow p' : q]$, and by pull-out $x_0 : p'[\emptyset \rightarrow q]$. That is, $p : q \in (p, \emptyset)^*$, which contradicts our assumption.

Therefore, there exists at least one A_i , $1 \leq i \leq n$, such that $p : q : A_i \notin (p, \emptyset)^*$. By inductive hypothesis the values of $p : q : A_i$ are distinct and therefore the value of $p : q$ is distinct. \square

Observation 3.5 *Let p, pq be paths, and v a value returned by $\text{newRow}(p, (p', \emptyset)^*)$, where p' is a prefix of p . If $q(v)$ is not a value distinct from any other value in I then there exists a prefix q' of q such that $p : q' \in (p', \emptyset)^*$.*

Proof. It is a direct consequence of Observation 3.4. \square

Observation 3.6 *Let $p, p : q$ be paths such that $p \notin \text{closure}$, type of p is $\langle A_1, \dots, A_k \rangle$, and for all A_i , $1 \leq i \leq k$, $p : A_i \in \text{closure}$. If $p : q \in (p, \emptyset)^*$ then $p : q \in \text{closure}$.*

Proof. Let $p \equiv x_0 : p'$. If $|q| = 1$ then it is direct because by assumption for all A_i , $x_0 : [X \rightarrow p' : A_i]$. Suppose $|q| > 1$, i.e., $q = A_1^i : q'$, where, $|q'| \geq 1$. By assumption, $x_0 : p'[\emptyset \rightarrow q]$. Then by push-in $x_0[p' \rightarrow p' : A_1^i : q']$. By full-locality, $x_0[p' : A_1^i \rightarrow p' : A_1^i : q']$, and then by transitivity, $x_0[X \rightarrow p' : A_1^i : q']$. Therefore, $p : q \in \text{closure}$. \square

Observation 3.7 *Let $p, p : q$ be paths, v a value of $p(I)$, and v_1, v_2 two elements in v such that $v_1 \neq v_2$. If $q(v_1) = q(v_2)$ then there exists a prefix q' of q such that $p : q' \in \text{closure}$.*

Proof. Let $q \equiv A_1 : \dots : A_k$. Suppose, by contradiction, that for all prefix q'' of q , $p : q'' \notin \text{closure}$. By Observation 3.2, if v has at has two distinct elements then either:

1. p is not in *closure* and $A_1 \in \text{closure}$: in this case, there exists a prefix, namely A_1 , such that $p : A_1$ is in *closure*;
2. $p \equiv x_0$, or $p \in \text{closure}$: x_0 is always built by *assignVal*, and by Observation 3.3 v was built either by *newRow*, or *assignVal*. By construction, v cannot be built by *newRow*, because this function always returns a singleton set. Therefore, v was built by *assignVal*. If there is no prefix q'' of q such that $p : q''$ is in *closure*, $q''(v_1)$ and $q''(v_2)$ were built by *computeNew*. But by Observation 3.4 these values are distinct, which contradicts our assumption that $q(v_1) = q(v_2)$. Therefore, there exists a prefix q' of q such that $p : q' \in \text{closure}$. \square

Now, we're ready to prove the completeness of the inference rules.

Lemma 3.2 *The NFD-rules are complete for all instances that contain no empty sets.*

Proof. From the definition of closure, $x_0 : [X \rightarrow y]$ follows from a given set of NFDs Σ using the NFD-rules if and only if $x_0 : y \in (x_0, X, \Sigma)^{(*,SC)}$.

We have to show that considering the instance I constructed by the algorithm in Figure 3.1:

1. $I \models \Sigma$
2. $I \not\models x_0 : [X \rightarrow y]$ if $x_0 : y \notin (x_0, X, \Sigma)^{(*,SC)}$.

1) $I \models \Sigma$

We will show that for any $f \equiv u_0 : [U \rightarrow z] \in \Sigma$, $I \models f$. Suppose otherwise, that $I \not\models f$.

If x_0 is not prefix of $u_0 : z$ then there exists a single value for $u_0 : z(I)$ and therefore I cannot violate f . Therefore, if $I \not\models f$ then x_0 is a prefix of $u_0 : z$.

Suppose $|u_0| < |x_0|$, i.e., $x_0 = u_0 : u'_0$. Let $f \equiv u_0 : [B_1u_1, \dots, B_lu_l, u'_0 : u_{l+1}, \dots, u'_0 : u_k \rightarrow u'_0 : z]$, where u'_0 is not a prefix of any B_iu_i , $1 \leq i \leq l$. Applying the prefix rule multiple times we have $u_0 : [B_1, \dots, B_l, u'_0 : u_{l+1}, \dots, u'_0 : u_k \rightarrow u'_0 : z]$. Applying the locality, and pull-out rules, we get $f' \equiv u_0 : u'_0 : [u_{l+1}, \dots, u_k \rightarrow z]$. Since there exists a unique value for all $u_0 : B_iu_i(I)$, $1 \leq i \leq l$, if $I \models f'$ then $I \models f$.

Therefore, we can assume that $|u_0| \geq |x_0|$, i.e. $u_0 \equiv x_0 : u'_0$. Let w be the largest common prefix between z and any $u \in U$. We will show that $I \models f$ by induction on $|w|$.

Base Case: $|w| = 0$

Let $f \equiv u_0 : [u_1, \dots, u_m \rightarrow z]$, and v an arbitrary value of $u_0(I)$. Suppose that $u_0 : z \in \text{closure}$. If $I \not\models f$, there must exist two distinct values of $z(v)$. Therefore, there exists a prefix z' of z not in *closure*, such that u_0z' was built by *newRow*, and $u_0 : z \notin (u_0z', \emptyset)^*$. Since $|w| = 0$, by full-locality rule, for all prefix z' of z $u_0 : [z' \rightarrow z]$. Let $z \equiv z'z''$. By

pull-out rule, $u_0 : z'[\emptyset \rightarrow z'']$ for all prefix z' of z . That is, there is no prefix z' of z such that $u_0 : z \notin (u_0 z', \emptyset)^*$. Therefore if $u_0 : z \in \text{closure}$ then all elements in v agree on their values of $z(v)$, and $I \models f$.

Now consider that $u_0 : z \notin \text{closure}$. We will first show that if $u_0 : z \notin \text{closure}$ then for any element $e \in v$ there exists a single value of $z(e)$. Given this, we will show that these values are distinct. Suppose, by contradiction, that $z(e)$ contains more than one value. If for all prefix p of z , $u_0 : p \notin \text{closure}$ then, by their construction using function *computeNew*, there exists a unique value of $z(e)$. Therefore, if there exists two different values of $z(e)$ then there exists a prefix z' of z such that $u_0 : z' \in \text{closure}$. Let $z \equiv z' : z''$, and $u_0 \equiv x_0 : u'_0$. Since there is no common prefix between z and any $u_i \in U$, by the full-locality rule $x_0 : u'_0[z' \rightarrow z' : z'']$, by pull-out, $x_0 : u'_0 : z'[\emptyset \rightarrow z'']$, and by push-in $x_0 : [u'_0 : z' \rightarrow u'_0 : z' : z'']$. Then, by transitivity, $u_0 : z \in \text{closure}$, which contradicts our assumption. Therefore, if $u_0 : z \notin \text{closure}$ then there exists no prefix z' of z such that $u_0 : z' \in \text{closure}$ and for any element $e \in u_0(I)$ there exists a single value of $z(e)$.

Now we have to show that there are no two distinct elements v_1, v_2 in $u_0(I)$ such that for all $u_i \in U$ $u_i(v_1) = u_i(v_2)$, and $z(v_1) \neq z(v_2)$. Suppose, by contradiction, that such elements exist. By construction, either $u_0 \notin \text{closure}$ and for all element attributes $u_0 : A$ of u_0 , $u_0 : A \in \text{closure}$, or $u_0 \in \text{closure}$ (or $u_0 \equiv x_0$) and there exists an attribute $u_0 : A \notin \text{closure}$. Consider the first case. Observe that there exists an element attribute A of u_0 such that A is a prefix of z . But we have shown that for all prefix z' of z , $u_0 : z' \notin \text{closure}$. Therefore, if $u_0(I)$ has two elements it must be the case that $u_0 \in \text{closure}$ or $u_0 \equiv x_0$. By Observation 3.7, if for all $u_i \in U$, $u_i(v_1) = u_i(v_2)$ then for all u_i there exists a prefix u'_i such that $u'_i \in \text{closure}$. Since there is no common prefix between any $u_i \in U$ and z , applying the prefix rule we get $x_0 : u'_0 : [u'_1, \dots, u'_n \rightarrow z]$, and by push-in, we get $x_0 : [u'_0, u'_0 : u'_1, \dots, u'_0 : u'_n \rightarrow u'_0 : z]$. Since $u_0 \in \text{closure}$ or $u_0 \equiv x_0$, by transitivity, $u_0 : z \in \text{closure}$, a contradiction. Therefore, there are no two elements v_1, v_2 that agree on all the u_i 's and disagree on z .

Inductive Step: $|w| > 0$.

Let $w \equiv A : w'$, and $f \equiv u_0 : [A : u_1, \dots, A : u_k, u_{k+1}, \dots, u_m \rightarrow A : z]$, where A is not

prefix of any u_i , $k < i \leq m$. By the locality rule, $u_0 : A[u_1 \dots u_k \rightarrow z]$. By inductive hypothesis, this NFD is satisfied. Therefore, if for every value v of $u_0(I)$ all elements agree on the value of A , then I cannot violate f . So, there exists a value v of $u_0(I)$ such that v has at least two elements, v_1, v_2 , and $A(v_1) \neq A(v_2)$.

We will first show that if there exist two elements $v_1, v_2 \in u_0(I)$ such that $A(v_1) \neq A(v_2)$ and for all u_i , $1 \leq i \leq m$, $u_i(v_1) = u_i(v_2)$ then $u_0 : z \in \text{closure}$. Given this, we will show that $z(v_1) = z(v_2)$.

Let $f \equiv u_0[u_1, \dots, u_m \rightarrow z]$, where $z \equiv A : z'$, and $u_0 \equiv x_0 : u'_0$. By Observation 3.7, if $u_i(v_1) = u_i(v_2)$ for all u_i , $1 \leq i \leq m$, then there exists a prefix u'_i of u_i , such that $u_0 : u'_i \in \text{closure}$. Let u'_i be the longest prefix of u_i such that $u_0 : u'_i \in \text{closure}$. We will consider two cases:

Case 1: For all u_i , u'_i is not a prefix of z .

In this case, we can apply the prefix rule multiple times and get $u_0 : [u'_1, \dots, u'_m \rightarrow z]$. By push-in we get $f' \equiv x_0 : [u'_0, u'_0 : u'_1, \dots, u'_0 : u'_m \rightarrow u'_0 : z]$.

The case when $x_0 : u'_0 \in \text{closure}$ or $u_0 \equiv x_0$ is direct: by transitivity with f' , $u_0 : z \in \text{closure}$. Consider the case when $u_0 \notin \text{closure}$. Since $v = u_0(I)$ is not a singleton set, by construction, $u_0 : A \in \text{closure}$, $v = \{v_1, v_2\}$, and either v_1 or v_2 was built by $\text{newRow}(u_0, (u_0, \emptyset)^*)$. Moreover, since $A(v_1) \neq A(v_2)$, $u_0 : A \notin (u_0, \emptyset)^*$. Since for all u_i , $1 \leq i \leq m$, $u_i(v_1) = u_i(v_2)$, by Observation 3.5 for all u_i there exists a prefix u'_i , such that $u_0 : u'_i \in (u_0, \emptyset)^*$. Then by transitivity, $u_0 : z \in (u_0, \emptyset)^*$. That is, $x_0 : u'_0[\emptyset \rightarrow z]$. By push-in rule $x_0 : [u'_0 \rightarrow u'_0 : z]$. Recall that $z \equiv A : z'$. Then by full-locality rule $x_0 : [u'_0 : A \rightarrow u'_0 : A : z']$. Since $u_0 : A \in \text{closure}$, by transitivity, $u_0 : z \in \text{closure}$.

Case 2: There exists a u_i such that u'_i is a prefix of z .

If there exists a $u'_i \in \text{closure}$ such that $u'_i = z$, then by the reflexivity rule $u_0 : z \in \text{closure}$. Let p' be the largest prefix among the u'_i 's such that $|p'| < |z|$. That is, $f \equiv x_0 : u'_0 : [p' : p_1, \dots, p' : p_k, p_{k+1}, \dots, p_m \rightarrow p' : z']$, where for all p_i , $k < i \leq m$, p' is not a prefix of p_i . By Observation 3.7, since $v_1 \neq v_2$, for each p_i , $1 \leq i \leq k$, there exists a prefix p'_i of p_i such that $u_0 : p' : p'_i \in \text{closure}$. Then by locality, we can get $x_0 : u'_0 : p' : [p_1, \dots, p_k \rightarrow z']$, and

by prefix, $x_0 : u'_0 : p' : [p'_1, \dots, p'_k \rightarrow z']$. By push-in, $x_0 : [u'_0 : p', u'_0 : p' : p'_1, \dots, u'_0 : p' : p'_k \rightarrow u'_0 : p' : z']$. Therefore, we can conclude that $x_0 : u'_0 : p' : z' \equiv u_0 : z \in \text{closure}$.

Now, we will show that if $u_0 : z \in \text{closure}$ then $z(v_1) = z(v_2)$. Suppose not. Recall that $f \equiv u_0 : [u_1, \dots, u_m \rightarrow z]$, and w is the largest common prefix between z and any u_i . Let $w \equiv A : w'$, v be a value of $u_0(I)$, and v_1, v_2 two elements in v such that $v_1(A) \neq v_2(A)$, and for all u_i , $1 \leq i \leq m$, $u_i(v_1) = u_i(v_2)$, but $z(v_1) \neq z(v_2)$.

Since $z(v_1) \neq z(v_2)$, and $u_0 : z \in \text{closure}$, it must be the case that either $z(v_1)$, or $z(v_2)$, or both were built by $\text{newRow}(u_0 : z, (p, \emptyset)^*)$. We will first argue that $|p| \geq |u_0|$. Suppose not. Then $u_0 \equiv p : p'$, and $z(v_1)$ was built by $\text{newRow}(p : p' : z, (p, \emptyset)^*)$. By construction, $p : p'$ was also built by $\text{newRow}(p : p', (p, \emptyset)^*)$, and since $|p'| \geq 1$, $p : p'(I) \equiv u_0(I)$ is a singleton set, a contradiction. Therefore, $|p| \geq |u_0|$.

Let $u_0 : z \equiv p : z'$. Consider the case when $|p| > |w|$. Then by full-locality and pull-out rules, $p[\emptyset \rightarrow z']$, and $u_0 : z \in (p, \emptyset)^*$. By construction, $z(v_1) = z(v_2) = \text{value}(z)$.

Therefore, if $z(v_1) \neq z(v_2)$ it must be the case that $|u_0| \leq |p| \leq |w|$. Let $p \equiv u_0 : p'$, $w \equiv p' : w'$, and $f \equiv u_0 : [p' : u_1, \dots, p' : u_k, u_{k+1}, \dots, u_m \rightarrow p' : z']$. By full-locality and pull-out rules, $p : [u_1, \dots, u_k \rightarrow z']$. Since for all u_i , $1 \leq i \leq k$, $p' : u_i(v_1) = p' : u_i(v_2)$, by Observation 3.5, there exists a prefix u' of u_i such that $p : u'_i \in (p, \emptyset)^*$. If $p' : z'(v_1)$ was built by $\text{newRow}(p : z', (p, \emptyset)^*)$ then for all prefix z'' of z' , $p : z'' \notin (p, \emptyset)^*$. But since $p : u'_i$ is not a prefix of $p : z'$, we can apply the prefix rule multiple times to get $p : [u'_1, \dots, u'_k \rightarrow z']$. By transitivity, $p : z' \in (p, \emptyset)^*$, a contradiction. Therefore, $z(v_1)$ (and $z(v_2)$) could not be built by newRow and $z(v_1) = z(v_2) = \text{value}(u_0 : z)$.

2) $I \not\equiv x_0 : [X \rightarrow y]$ if $x_0 : y \notin (x_0, X, \Sigma)^*, SC$

First, we will show that there exists a prefix p of $x_0 : y$ such that $|p| \geq |x_0|$, and $p(I)$ has two elements, v_1 , and v_2 . We will then establish that v_1 and v_2 coincide in all attributes in X but not in y if $y \notin \text{closure}$.

Suppose, by contradiction, that in I , the value of all prefix p of y is a singleton set. Start by considering $p = \epsilon$. Then $p = x_0$, and by construction, $x_0(I)$ was built by assignVal . If it contains a single element then for all element attributes A of x_0 , $x_0 : A \in \text{closure}$, and

the value of $x_0 : A$ is given by $assignVal(x_0 : A)$. The same argument can be used for the value of each attribute $x_0 : A$ built by $assignVal(x_0 : A)$: it has only one element if all element attributes of $x_0 : A$ are in $closure$, and built by $assignVal$. But, by assumption, $x_0 : y \notin closure$. Therefore, there exists a prefix p of y such that $x_0 : p(I)$ was built by $assignVal$ and contains two elements.

Let p be the largest prefix of y that satisfies this condition, and $f = x_0 : [p : u_1, \dots, p : u_k, u_{k+1}, \dots, u_m \rightarrow p : y']$. We claim that if $I \not\models x_0 : p : [u_1, \dots, u_k \rightarrow y']$, then $I \not\models f$. If $I \models x_0 : p : [u_1, \dots, u_k \rightarrow y']$ then there exists a value v' of $x_0(I)$, and v of $p(v')$, with two elements v_1, v_2 such that $u_i(v_1) = u_i(v_2)$ for all i , $1 \leq i \leq k$, and $y'(v_1) \neq y'(v_2)$. Since for all u_i , $k < i \leq m$, p is not a prefix of u_i , there exist values of $u_i(v')$ that coincide. Therefore, v' is a witness that $I \not\models f$.

We only have to show that such $x_0 : p(I) = v = \{v_1, v_2\}$ exists. By the reflexivity rule, all $x_0 : p : u_i$, $1 \leq i \leq k$, are in $closure$. Recall that v was built by $assignVal$. Therefore, all element attributes of $x_0 : p$ were either built by $assignVal$ or $computeNew$, and both functions assign $value(p')$ to an attribute whenever $p' \in closure$. Hence, there exist values of $u_i(v_1), u_i(v_2)$, such that $u_i(v_1) = u_i(v_2) = value(x_0 : p : u_i)$. Now we have to show that $y'(v_1) \neq y'(v_2)$. Let $y' \equiv A_1 : \dots : A_n$. Since p is the largest prefix of $x_0 : y$ built by $assignVal$, for all A_i , $1 \leq i \leq n$, $A_1 : \dots : A_i \notin closure$, and $A_1 : \dots : A_i(v_1), A_1 : \dots : A_i(v_2)$ were built by $computeNew$. Given that $computeNew$ never returns the same value for a given attribute, $y'(v_1) \neq y'(v_2)$. \square

3.2 The Problem of Empty Sets

As mentioned earlier, the presence of empty sets causes difficulties in reasoning since formulas such as $\forall x \in R. P(x)$ are trivially true when R is empty. In particular, the transitivity rule is no longer sound in the presence of empty sets, as illustrated below.

Example 3.4 The instance of R below satisfies $R : [A \rightarrow B : C]$, $R : [B : C \rightarrow D]$, but not $R : [A \rightarrow D]$.

A	B	D	E
1	\emptyset	2	3
1	\emptyset	3	4
2	$\{<C : 3>\}$	4	5

□

One reasonable solution to this problem is to disallow empty sets only in certain portions of the schema; this is analogous to specifying NON-NULL for certain attributes in a relational schema. We use $p \neq \emptyset$ to denote that p is a path where empty sets are known not to occur. The inference rules can then be modified to make use of this information.

Before presenting a new transitivity rule, we introduce a new relation *follow* between paths.

Definition 3.5 *Let $p_1 \equiv p'_1 A$ be a path expression. We say that p_1 follows p_2 if p'_1 is a proper prefix of p_2 .*

Intuitively, p_1 follows p_2 if p_1 only traverses the set-valued attributes traversed by p_2 . That is, if $p_1 \equiv A_1 : \dots : A_{n-1} : A_n$, all labels A_i , except the last, denote set-valued attributes. Therefore, if $p_2 \equiv A_1 : \dots : A_{n-1} : p'_2$ then all set-valued attributes traversed by p_1 are also traversed by p_2 . For example, path $A : B$ follows $A : C$, $A : C : D$, but not A , E , and $F : G$. In particular, a path A follows any path p with $|p| \geq 1$, since $A \equiv \epsilon A$, and ϵ is a proper prefix of any path containing at least one label.

The new transitivity rule is then defined as:

$$\frac{x_0 : [X \rightarrow x_1], \dots, x_0 : [X \rightarrow x_n], x_0 : [x_1, \dots, x_n \rightarrow yA], \quad \forall p \in \{x_1, \dots, x_n\} - X, p \text{ does not follow } y \Rightarrow p \neq \emptyset}{x_0 : [X \rightarrow y]} \quad (\text{transitivity-with-empty-sets})$$

The fact that transitivity does not generally hold in the presence of empty sets has also influenced our definition of NFDs to allow only single paths on the right-hand side rather than a set of paths.

Recall that in the relational model, a functional dependency (FD) $X \rightarrow Y$, where X, Y are sets of attributes, can be decomposed into a set of FDs with single attributes on the right-hand side of the implication. Unfortunately, the decomposition rule follows from reflexivity and transitivity and cannot therefore be uniformly applied with NFDs in the presence of empty sets, as illustrated by the next example.

Example 3.5 The instance I of R below satisfy $R[A \rightarrow B : C, E : F]$, but not $R[A \rightarrow E : F]$

A	B		E	
	C	D	F	G
1	2	3	4	5
1	\emptyset		3	4

□

This motivates us to define NFDs with a single consequence instead of the more general form, which can be counter-intuitive in the presence of empty sets.

The presence of empty sets also affects the prefix rule. Consider the instance I presented in Example 3.2. Notice that I satisfies $R : [B : C \rightarrow E]$, but not $R : [B \rightarrow E]$. A modified prefix rule to take this into account is:

$$\frac{x_0 : [x_1 : A, x_2, \dots, x_k \rightarrow y], \quad x_1 \text{ has one or more labels, } x_1 \text{ is not prefix of } y, \quad x_1 \neq \emptyset \quad (\text{prefix-with-empty-sets})}{x_0 : [x_1, x_2, \dots, x_k \rightarrow y]}$$

Generalize the inference rules for NFDs to the case where the user defines which set-valued paths are known to have at least one element would allow us to reason about constraints for a larger family of instances. We believe this is natural requirement to make, since definition of cardinality has long been recognized as integral part of schema design[Che76].

3.3 Simple NFDs

Notice that **push-in** and **pull-out** rules simply change between equivalent forms of NFDs. I.e., NFDs of the form $R : y : [x_1, \dots, x_k \rightarrow z]$ are equivalent to $R : [y, y : x_1, \dots, y : x_k \rightarrow y : z]$. Therefore, we could change the definition of NFDs to allow only relation names as base paths (x_0), without changing its expressive power.

In this simpler form of NFDs it can be shown that there are only 6 inference rules: push-in and pull-out are unnecessary. Of the remaining rules, only the locality must be modified to that of full-locality:

$$\frac{x_0 : [x : X, Y \rightarrow x : z], \quad x \text{ is not a proper prefix of any } y \in Y}{x_0 : [x, x : X \rightarrow x : z]} \quad (\text{full-locality})$$

Note that full-locality “combines” the locality and push-in rules. Since this simple form of NFDs appears to closely resemble the definition of functional dependencies for the relational model, the natural question that arises is: Can we infer all the simple NFDs using only the Armstrong axioms and the inference rules for multivalued dependencies?

The answer to this question is no. In order to establish this result, we will first define join and multivalued dependencies (MVDs), along with a decision procedure for determining logical implication based on tableaux. Then, the construction of a flattened representation of a nested relation based on successive unnest operations is presented. This representation is lossless in the sense that the nested relation can always be reconstructed by a sequence of nest operations. It also satisfies a set of multivalued dependencies. Given that NFDs can also be mapped to FDs defined on the flattened relation, we can then show that some NFDs cannot be inferred using only rules for FDs+MVDs.

3.3.1 Join and Multivalued Dependencies (MVDs)

Before² defining a join dependency, let us recall the definition of natural join. Given a relation schema $S(R)$ with set of attributes $\mathcal{A}(R)$, sets $X_1, \dots, X_n \subseteq \mathcal{A}(R)$, and instances

²This section is based on [AHV95].

A	B	C	D
1	2	5	7
1	2	6	7
1	2	5	8
1	2	6	8
1	3	5	4
1	3	6	4

 $=$

A	B
1	2
1	3

 \bowtie

A	C
1	5
1	6

 \bowtie

B	D
2	7
2	8
3	4

Figure 3.2: A relation that satisfies $\bowtie [AB, AC, BD]$.

I_j over X_j , $j \in [1, n]$, the natural join of the I_j 's is the set of tuples s defined on the set of attributes $X_1 \cup \dots \cup X_n$, generated by concatenating tuples in the I_j relations that agree on the common attributes. Formally,

$$\bowtie_{j=1}^n I_j = \{s \text{ over } \cup X_j \mid \Pi_{X_j}(s) \in I_j \text{ for each } j \in [1, n]\}.$$

A join dependency over a set of attribute sets is satisfied by a relation r if it is equal to the join of the projections of subsets of its attributes.

Definition 3.6 A join dependency (JD) over attribute set $\mathcal{A}(R)$ is an expression of the form $\bowtie [X_1, \dots, X_n]$, where $X_1, \dots, X_n \subseteq \mathcal{A}(R)$ and $\cup_{i=1}^n X_i = \mathcal{A}(R)$. A relation r over $S(R)$ satisfies $\bowtie [X_1, \dots, X_n]$ if $r = \bowtie_{j=1}^n \Pi_{X_j}(r)$.

As an example, the relation r in Figure 3.2 satisfies the JD $\bowtie [AB, AC, BD]$, since $r = \Pi_{AB}(r) \bowtie \Pi_{AC}(r) \bowtie \Pi_{BD}(r)$.

A multivalued dependency is a special case of a join dependency when the number of attribute sets involved in the JD is two.

Definition 3.7 A multivalued dependency (MVD) over $S(R)$ is an expression of the form $R : [X \twoheadrightarrow Y]$, where $X, Y \subseteq \mathcal{A}(R)$. A relation r over $S(R)$ satisfies $R : [X \twoheadrightarrow Y]$ if r satisfies the join dependency $\bowtie [XY, X(\mathcal{A}(R) - Y)]$.

Intuitively, a multivalued dependency, $R : [X \twoheadrightarrow Y]$ states that given a value for the

$\frac{R : [X \twoheadrightarrow Y]}{R : [X \twoheadrightarrow (\mathcal{A}(R) - X - Y)]}$	(complementation)
$\frac{R : [X \twoheadrightarrow Y], \quad V \subseteq W}{R : [WX \twoheadrightarrow VY]}$	(augmentation for MVDs)
$\frac{R : [X \twoheadrightarrow Y], \quad R : [Y \twoheadrightarrow Z]}{R : [X \twoheadrightarrow (Z - Y)]}$	(transitivity for MVDs)
$\frac{R : [X \rightarrow Y]}{R : [X \twoheadrightarrow Y]}$	(conversion)
$\frac{R : [X \twoheadrightarrow Y], \quad Z \subseteq Y, \quad \text{for some } W \text{ disjoint from } Y \quad R : [W \rightarrow Z]}{R : [X \twoheadrightarrow Z]}$	(interaction)

Table 3.3: Rules for FD+MVD implication

attributes of X there is a set of zero or more associated values for the attributes of Y , and this set of Y -values is not connected in any way to values of the attributes in $\mathcal{A}(R) - X - Y$.

The inference rules for functional and multivalued dependencies considered together consist of the Armstrong Axioms combined with the inference rules illustrated in Table 3.3.

Although there exists a set of inference rules for MVDs and FDs considered together that are proven to be sound and complete, there is no axiomatization for JDs [AHV95]. However, there exists a decision procedure for determining logical implication of JDs based on tableaux, as we present next.

A *tableau*³ over a relation schema $S(R)$ is a set of rows, pictured as a matrix, with one column for each attribute in the attribute set $\mathcal{A}(R)$. The rows of the matrix are composed of variables. A *tableau query* is a pair (T, u) , where T is a tableau and u is a free tuple called the *summary* of the tableau query. Variables in u that also occur in T , are called *distinguished variables*, denoted by subscripted a 's. Remaining variables in T are called *nondistinguished variables*, denoted by subscripted b 's. The summary tuple u represents the tuples included in the answer to the query.

Definition 3.8 *Let $\sigma = \bowtie [X_1, \dots, X_n]$ be a JD. The tableau query for σ is (T_σ, t) , where*

³This discussion on tableaux is based on [AHV95] and [MMS79].

		A	B	C	D
T_σ	t_1	a_1	a_2	b_1	b_2
	t_2	a_1	b_3	a_3	b_4
	t_3	b_5	a_2	b_6	a_4
	t	a_1	a_2	a_3	a_4

Figure 3.3: A tableau query (T_σ, t) , where $\sigma = \bowtie [AB, AC, BD]$.

for some t_1, \dots, t_n :

- t is a free tuple over R with a distinct variable for each attribute,
- $T_\sigma = \{t_1, \dots, t_n\}$
- $\pi_{X_i}(t_i) = \pi_{X_i}(t)$ for $i \in [1, n]$, and
- the other attributes of the t_i 's hold distinct variables.

As an example, the tableau query (T_σ, t) , where $\sigma = \bowtie [AB, AC, BD]$, is illustrated in Figure 3.3.

Given a set of dependencies Σ , the chase is a general technique that can be used to transform a tableau query q into a query q' such that q and q' are equivalent with respect to Σ . That is, given any instance I such that I satisfies Σ , the results of applying q and q' into I are the same.

Let (T, t) be a tableau query over relation schema $S(R)$. If Σ is a set of JDs, the chase is based on the successive application of the following rule:

jd-rule: Let $\tau = \bowtie [X_1, \dots, X_n]$ be a JD over $S(R)$, and let u be a free tuple over $S(R)$ not in T , and suppose $u_1, \dots, u_n \in T$ satisfy $\pi_{X_i}(u_i) = \pi_{X_i}(u)$ for $i \in [1, n]$. Then the *result of applying* the JD τ to (u_1, \dots, u_n) in (T, t) is the tableau query $(T \cup u, t)$.

As an example, let $\tau = \bowtie [AB, BCD]$. Then applying τ to (t_1, t_3) in the tableau of Figure 3.3 results in the tableau illustrated in Figure 3.4(a).

		A	B	C	D
	t_1	a_1	a_2	b_1	b_2
T_σ	t_2	a_1	b_3	a_3	b_4
	t_3	b_5	a_2	b_6	a_4
	t_4	a_1	a_2	b_6	a_4
t		a_1	a_2	a_3	a_4

		A	B	C	D
	t_1	a_1	a_2	b_1	b_2
T_σ	t_2	a_1	b_3	a_3	b_4
	t_3	b_5	a_2	b_6	a_4
	t_4	a_1	a_2	b_6	a_4
	t_5	a_1	a_2	a_3	a_4
t		a_1	a_2	a_3	a_4

(a) Result of applying $\bowtie [AB, BCD]$ to (t_1, t_3) . (b) Result of applying $\bowtie [ABD, AC]$ to (t_4, t_2) .

Figure 3.4: Application of a jd-rule

Let Σ be a set of JDs. A *chasing sequence* of (T, t) by Σ is a sequence

$$(T, t) = (T_0, t_0), \dots, (T_n, t_n)$$

such that for each $i \geq 0$, (T_{i+1}, t_{i+1}) is the result of applying some JD in Σ to (T_i, t_i) . The sequence is *terminal* if it is finite and no dependency in Σ can be applied to it. The last element of the terminal sequence is denoted as $chase(T, t, \Sigma)$. The *length* of the chasing sequence is the number of tableaux in the sequence, that is, the number of times a jd-rule has been applied.

It has been shown that chasing the tableau representation of a JD is an effective way for determining their logical implication, as the following theorem states.

Theorem 3.1 [AHV95] *Let Σ and $\{\sigma\}$ be sets of JDs over relation schema $S(R)$, and let (T_σ, t_σ) be the tableau query of σ , and T be the tableau in $chase(T_\sigma, t_\sigma, \Sigma)$. Then Σ logically implies σ if and only if $t_\sigma \in T$.*

That is, if by chasing (T_σ, t_σ) using Σ , a line composed of only distinguished variables is produced, then Σ implies σ . In the example illustrated in Figure 3.4, it is true that $\{\bowtie [AB, BCD], \bowtie [ABD, AC]\}$ logically implies $\sigma = \bowtie [AB, AC, BD]$, since a line $(A : a_1, B : a_2, C : a_3, D : a_4)$ is produced by chasing (T_σ, t_σ) .

The results presented in this section will be useful in proving some properties of the flattened representation of a nested relation, that will be discussed in the next section.

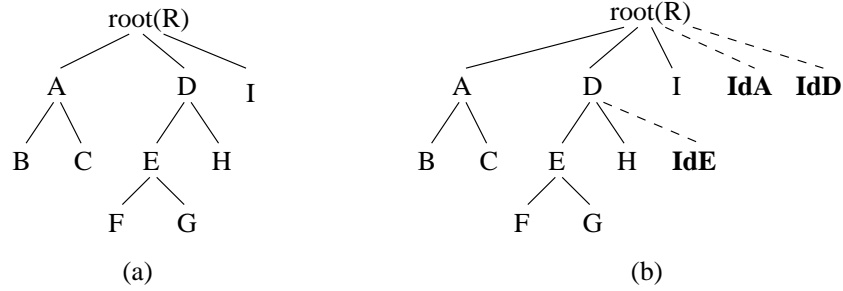


Figure 3.5: A schema tree and its extension with set-identifier attributes

3.3.2 Construction of the Flattened Representation of a Nested Relation

Having defined join and multi-valued dependencies, we are now going to present how a lossless flattened representation of a nested relation can be built, and how the hierarchical structure of the nested relation can be captured by a join dependency or a set of multi-valued dependencies. The approach for flattening a nested relation is similar to the one described in [LS97], in which every set-valued attribute is replaced with a fresh atomic value, that we call an *identifier*.

The schema of a nested relation $S(R)$ can be represented as a tree, called the *schema tree* of R and denoted as $Tree(R)$. In a schema tree, leaf nodes correspond to base type attributes in $S(R)$. We denote this set of nodes as $Leaves(R)$. The set of internal nodes, excluding the root node, correspond to attributes of set type, denoted as $IntNodes(R)$. Given a node n in the tree, we denote by $Child(n)$ the set of attributes that are children of n in $Tree(R)$.

As an example, let $S(R) = \{ \langle A : \{ \langle B, C \rangle \}, D : \{ \langle E : \{ F, G \}, H \rangle \}, I \rangle \}$. The schema tree $Tree(R)$ is illustrated in Figure 3.5(a). In $Tree(R)$, $Leaves(R) = \{ B, C, F, G, H, I \}$, $IntNodes(R) = \{ A, D, E \}$, and $Child(D) = \{ E, H \}$.

The construction of the flattened representation of a nested relation involves the operations *nest* and *unnest* presented next. Let r be a nested relation of schema $S(R)$.

Definition 3.9 *Let r' be the result of the nest operation $nest_{A=X}(r)$. The schema of r' , represented as a tree $Tree(R')$, is obtained from $Tree(R)$, by inserting a new child A under*

the root node, $\text{root}(R')$, and making all attributes in X children of A in $\text{Tree}(R')$. A tuple t' is in r' if and only if there exists a tuple $t \in r$ such that:

1. for all $a \in (\text{Child}(\text{root}(R)) - X)$, $t.a = t'.a$
2. $t'.A = \Pi_X(\{v \in r \mid \text{for all } a \in (\text{Child}(\text{root}(R)) - X), v.a = t'.a\})$

where $t.a$ denotes the value of attribute a in a tuple t .

Definition 3.10 Let r' be the result of the unnest operation $\text{unnest}_A(r)$. The schema of r' , represented as a tree $\text{Tree}(R')$, is obtained from $\text{Tree}(R)$ by removing the node A , and connecting all the children of A to the root node. A tuple t' is in r' if and only if there exists $t \in r$ such that:

1. for all $a \in \text{Child}(\text{root}(R) - \{A\})$, $t'.a = t.a$
2. there exists $v \in t.A$ such that for all $a \in \text{Child}(A)$, $t'.a = v.a$

Given this, we are ready to define how a flattened representation of a nested relation can be built. The construction involves two steps: first, the relation is extended to associate an identifier to each set-valued attribute, and then a sequence of unnest operations is applied to the resulting relation. The presence of set-identifiers in the flattened relation guarantees that the nested relation can be recreated from its flattened representation. The two steps are now presented in detail.

Let r be a nested relation of schema $S(R)$, and SetId a Skolem-function that generates an identifier for every distinct set value. First, we construct a relation r' from r by creating for each set-valued attribute A in $S(R)$, a corresponding identifier attribute Id_A . More precisely, if A in $\text{IntNodes}(R)$, and $A \in \text{Child}(A')$, then a new child Id_A is inserted under A' . In the relation r' , distinct values are assigned for each set value, using function SetId . Then, if in the resulting relation $R' : P_{\text{Id}_A}$, and $R' : P_A$ are path expressions corresponding to attributes Id_A , and A , respectively, and v_1 , and v_2 are two tuples in r' , $R' : P_{\text{Id}_A}(v_1) = R' : P_{\text{Id}_A}(v_2)$ if and only if $R' : P_A(v_1) = R' : P_A(v_2)$.

A		D			I	
B	C	E		H	9	
	1	2	F	G		7
		3	4			
		5	6			
		F		G	8	
		3	4			
B		E			H	10
1	2	F	G		8	
3	4	3	4			

(a)

A		D				I	Id_A	Id_D		
B	C	E		H	Id_E	9	i_1	i_2		
	1	2	F	G					i_5	
		3	4							
		5	6							
		F		G	8	i_6				
		3	4							
B		E			H	Id_E	10	i_3	i_4	
1	2	F	G			8				i_6
3	4	3	4							

(b)

Figure 3.6: A nested relation extended with set-identifier attributes

Id_A	B	C	Id_D	Id_E	F	G	H	I
i_1	1	2	i_2	i_5	3	4	7	9
i_1	1	2	i_2	i_5	5	6	7	9
i_1	1	2	i_2	i_6	3	4	8	9
i_3	1	2	i_4	i_6	3	4	8	10
i_3	3	4	i_4	i_6	3	4	8	10

Figure 3.7: Flattened representation of a nested relation

As an example, Figure 3.5(b) illustrates the tree schema of Figure 3.5(a) extended with identifier attributes, and Figure 3.6 illustrates an instance r and its extension with values for the identifier attributes.

A flattened representation of r' can then be obtained by a sequence of unnest operations. Since the unnest operation is commutative [JS82], the only restriction on the order of the application of these operations is that we may not unnest an attribute unless higher-level nestings over that attribute have been unnested. In the example of Figure 3.5 it is not legal to unnest E before unnesting D , but the order of unnesting A and D is irrelevant.

Following the example in Figure 3.6, the result of the unnesting sequence

$$\text{unnest}_A(\text{unnest}_E(\text{unnest}_D(r')))$$

is illustrated in Figure 3.7.

This flattened representation of a nested relation is lossless in the sense that the nested

relation can always be reconstructed from it by a sequence of nest operations. In order to reach a nested schema $S(R)$, represented as a tree $Tree(R)$, the nesting sequence should move bottom-up. That is, we may only create an attribute A by nesting a set of attributes X either if all attributes in X are of base type or have been created by prior nest operations. Moreover, after performing a nest operation $nest_{A=X}(r')$, the identifier attribute id_A should be projected out from the result before the next nest operation.

As an example, the original relation r can be reconstructed from the flattened relation r' illustrated in Figure 3.7 by the following sequence of operations. Here, $\Pi_{\setminus(Id_A)}(r)$ denotes the projection over r of all attributes except Id_A .

$$\Pi_{\setminus(Id_A)}(nest_{A=\{BC\}}(\Pi_{\setminus(Id_D)}(nest_{D=\{EH\}}(\Pi_{\setminus(Id_E)}(nest_{E=\{FG\}}(r'))))))))$$

Let $flat(r)$ denote the flattened representation of r of schema $S(R)$. Recall that for any node n in a tree representation of a schema $Tree(R)$, $Child(n)$ denotes the set of children of node n . It can be shown that the following join dependency, denoted as $jd(R)$, holds in $flat(r)$:

$$\bowtie [attName(Child(root(R))), \{attName(A \cup Child(A) \mid A \in IntNodes(R))\}]$$

where $attName$ is a function that given a set of attribute names in $S(R)$, returns the set of corresponding attribute names in its flattened representation. That is, if A is of set type then $attName(\{A\}) = \{Id_A\}$; otherwise, $attName(\{A\}) = \{A\}$. In the running example, the join dependency satisfied by r' is

$$\bowtie [\{Id_A, Id_D, I\}, \{Id_A, B, C\}, \{Id_D, Id_E, H\}, \{Id_E, F, G\}]$$

In the sequence, assume that $jd(R) = \bowtie [X_1, X_2, \dots, X_n]$, where $X_1 = attName(Child(root(R)))$, and $X_i = attName(A_i \cup Child(A_i))$, for $2 \leq i \leq n$. That is, each X_i , $i \geq 2$, is composed of a set-identifier attribute Id_{A_i} , denoted as $setAtt(X_i)$, and the attributes in the flattened relation that correspond to the elements of A_i in the original relation. We denote this set as $elemAtt(X_i)$. We will now show that the flattened representation of a nested relation built as described satisfies $jd(R)$.

Lemma 3.3 *Let r be a nested relation over schema $S(R)$. Then $flat(r)$ satisfies $jd(R)$.*

Proof. Let $setValue$ be the inverse function of $SetId$. That is, $setValue(i) = s$ if and only if $SetId(s) = i$. Define a function $value$ that given a tuple (v_1, \dots, v_m) returns a tuple (v'_1, \dots, v'_m) , where for each $v_i, v'_i = setValue(v_i)$ if v_i is a set identifier, and $v'_i = v_i$, otherwise. Let r be a nested relation over $S(R)$. Suppose, by contradiction, that $flat(r)$ does not satisfy $jd(R)$. Then, there exists a tuple t not in $flat(r)$ such that for every $X_i, 1 \leq i \leq n, t[X_i] = t_i[X_i]$ for some tuple t_i in $flat(r)$. Here, $t[X]$ represents the restriction of tuple t over attributes X .

Since tuples $t_i, 1 \leq i \leq n$ exist in $flat(r)$, from the construction of $flat(r)$, it must be the case that for all $i, i \geq 2, value(t_i[elemAtt(X_i)]) \in setValue(t_i[setAtt(X_i)])$. In particular, $value(t_1[X_1])$ must be a tuple in the original nested relation.

Since $t[X_i] = t_i[X_i]$ for every $i, 1 \leq i \leq n$, all tuples t_i must agree on all common attributes. Recall that for simplicity we have assumed that all attributes in the original nested relational schema have distinct names. Therefore, the only common attributes among the X_i 's are the set-identifier attributes. From the hierarchical structure of the nested schema and the definition of $jd(R)$, it is true that for every set-identifier attribute Id_A in the schema of $flat(r)$, there exists $X_i, i \geq 2$ such that $Id_A = setAtt(X_i)$; moreover, either Id_A is in X_1 or Id_A is in some $elemAtt(X_j), j \geq 2$, and $j \neq i$. Since every value t_i must be pairwise consistent, and must exist in some level of nesting of r , t must be in $flat(r)$, which contradicts our initial assumption. Therefore, $flat(r)$ satisfies $jd(R)$. \square

From the construction of $flat(r)$ it can also be verified that the following set of multivalued dependencies, denoted as $mvs(R)$ holds in $flat(r)$:

$$\{attName(A) \twoheadrightarrow attName(Child(A)) \mid A \in IntNodes(R)\}$$

In fact, a join dependency of the form of $jd(R)$ is a special case of a class of dependencies called *acyclic join dependencies*. It has been shown [BFMY83] that a join dependency is acyclic if and only if it is equivalent to a set of multivalued dependencies.

Next, we will show that $jd(R)$ and $mvs(R)$ are equivalent.

Lemma 3.4 *Let r be a nested relation of schema $S(R)$, and $S(R')$ the schema of $\text{flat}(r)$. Then the set of MVDs $\text{mvds}(R)$ defined on R' is equivalent to the join dependency $\text{jd}(R)$ defined on R' .*

Proof. Let $\text{jd}(R) = \bowtie [X_1, X_2, \dots, X_n]$. We will show that for any relation $r \circ S(R)$, $r \models \text{mvds}(R)$ if and only if $r \models \text{jd}(R)$. That is, using the chase technique, we will show that $\text{mvds}(R) \models \text{jd}(R)$, and $\text{jd}(R) \models \text{mvds}(R)$.

$\text{mvds}(R) \models \text{jd}(R)$: Assume that sets X_i are partially ordered such that if $\text{setAtt}(X_i) \in \text{elemAtt}(X_j)$ then $i > j$. In other words, in the schema tree of R , if A_i is a child of A_j , and they correspond to $\text{setAtt}(X_i)$, and $\text{setAtt}(X_j)$, respectively, then $i > j$.

Let $(T_{\text{jd}(R)}, t_{\text{jd}(R)})$ be the tableau query for $\text{jd}(R)$. We will show by induction on n , the number of attribute sets in $\text{jd}(R)$, that there exists a chase sequence using $\text{mvds}(R)$ that produces tuples s_1, \dots, s_n such that for all i , $s_i[W_i] = t_{\text{jd}(R)}[W_i]$, where $W_i = \cup_{j=1}^i X_j$. That is, in s_i all attributes in X_1, \dots, X_i contain distinguished variables. The base case, $i = 1$ is direct, since $X_1 = \text{attName}(\text{Child}(\text{root}(R)))$, and there exists a line in $(T_{\text{jd}(R)}, t_{\text{jd}(R)})$ with distinguished variables for $\text{attName}(\text{Child}(\text{root}(R)))$. Assume that the statement holds for $i < k$. We shall show that it also holds for $i = k$. From the construction of $\text{mvds}(R)$, if $X_k = \{A_k\} \cup \text{Child}(A_k)$, then there exists an $\text{mvd}_k = A_k \twoheadrightarrow \text{Child}(A_k)$ in $\text{mvds}(R)$. By induction hypothesis, there exists a line s_{k-1} in the chase sequence of $(T_{\text{jd}(R)}, t_{\text{jd}(R)})$ with all distinguished variables for X_1, \dots, X_{k-1} . Since the X_i 's are partially ordered, $A_k \in X_1 \cup \dots \cup X_{k-1}$, and therefore $s_{k-1}[A_k] = a_k$, where a_k is a distinguished variable. By construction, there exists a line l_k , where $l_k[A_k] = a_k$, and all attributes in $\text{Child}(A_k)$ also contain distinguished variables. Since $l_k[A_k] = s_{k-1}[A_k] = a_k$, the application of the jd -rule on l_k , and s_{k-1} will produce a line s_k in which all attributes in $X_1 \cup \dots \cup X_k$ have distinguished variables.

From the discussion above, we can conclude that there exists a chase sequence of length n resulting in T such that $t_{\text{jd}(R)} \in T$. Therefore, by Theorem 3.1, $\text{mvds}(R) \models \text{jd}(R)$.

$\text{jd}(R) \models \text{mvds}(R)$: Let $\text{jd}(R) = [X_1, \dots, X_n]$. Recall that by construction, for each i , $1 < i \leq n$ there exists a MVD $\text{mvd}_i = A_i \twoheadrightarrow \text{Child}(A_i)$ in $\text{mvds}(R)$. Let mvd_k be one

such arbitrary MVD, and (T_{mvd_k}, t_{mvd_k}) be the tableau query for mvd_k . Recall that T_{mvd_k} contains two lines, l_1, l_2 , where in l_1 only attributes A_k and $Child(A_k)$ have distinguished variables, and in l_2 , attributes in $Y = \cup_{i=1}^n (X_i) - X_k$ contain distinguished variables. By the construction of $jd(R)$, $A_k \in Y$. Therefore, applying the jd-rule using $jd(R)$ will produce a line with all attributes with distinguished variables. Therefore, $jd(R) \models mvd_k$. \square

The previous lemmas show that $mvs(R)$ captures the hierarchical structure of the nested relation. In the next section, we will use this result in showing that the set of inference rules for FDs+MVDs is not equivalent to the set of rules for NFDs.

3.3.3 Relationship between NFDs and FDs+MVDs

In this section, we will first present a simple translation of NFDs to FDs on the flattened schema. We will then show that there are NFDs that can be inferred using the NFD-rules, but cannot be inferred from the FDs and the MVDs defined as described in the previous section.

By defining a function $attPath$ for mapping path expressions in $S(R)$ to attribute names in the flattened schema $S(R')$, NFDs can be easily translated to FDs defined on $S(R')$. Given that there is a one-to-one correspondence between values in r and $flat(r)$, it is easy to see that given a simple NFD $nfd = R : [x_1, \dots, x_{m-1} \rightarrow x_m]$, r satisfies nfd if and only if $flat(r)$ satisfies the FD $attPath(x_1), \dots, attPath(x_{m-1}) \rightarrow attPath(x_m)$.

Given this, we are ready to show that the set of inference rules for NFDs is not equivalent to the set of inference rules for FDs + MVDs.

Theorem 3.2 *Let r be a nested relation of schema $S(R)$, and $S(R')$ the schema of $flat(r)$. The set of inference rules for NFDs defined on R is not equivalent to the set of inference rules for FDs + MVDs defined on R' .*

Proof. From the eight inference rules for NFDs, three mirror Armstrong Axioms, and two exchange between equivalent forms of NFDs. Therefore, we will focus on the remaining

three: prefix, full-locality, and singleton. In particular, we will show that the prefix and full-locality rules can be captured by the rules for FDs+MVDs, but the singleton rule cannot.

Without loss of generality, suppose $S(R) = \{ \langle x : \{ \langle y_1, \dots, y_k \rangle \}, z_1, \dots, z_n \rangle \}$, with a corresponding flattened schema $S(R') = \{ id_x, y_1, \dots, y_k, z_1, \dots, z_n \}$. According to the discussion in Section 3.3.2, the following MVD can be defined on $S(R')$:

mvd1: $id_x \twoheadrightarrow y_1, \dots, y_k$

full-locality: The full-locality rule states that:

if $R : [x : y_a, \dots, x : y_b, z_l, \dots, z_m \twoheadrightarrow x : y_c]$ and x is not a prefix of any z_i , $i \in [1, n]$

then $R : [x, x : y_a, \dots, x : y_b \twoheadrightarrow x : y_c]$,

where $\{y_a, \dots, y_b\} \subseteq \{y_1, \dots, y_k\}$ and $\{z_l, \dots, z_m\} \subseteq \{z_1, \dots, z_n\}$.

For any relation r of $S(R)$, r satisfies the premise if and only if $flat(r)$ satisfies the following FD:

fd1: $y_a, \dots, y_b, z_l, \dots, z_m \twoheadrightarrow y_c$.

We can prove that in R' , $id_x, y_a, \dots, y_b \twoheadrightarrow y_c$ is a consequence of *mvd1*, and *fd1* as follows.

1. $id_x \twoheadrightarrow z_1, \dots, z_n$ by complementation of *mvd1*.
2. $id_x, y_a, \dots, y_b \twoheadrightarrow y_a, \dots, y_b, z_1, \dots, z_n$ by MVD-augmentation of 1.
3. $y_a, \dots, y_b, z_1, \dots, z_n \twoheadrightarrow y_c$ by FD-augmentation of *fd1*.
4. $y_a, \dots, y_b, z_1, \dots, z_n \twoheadrightarrow y_c$ by conversion of 3.
5. $id_x, y_a, \dots, y_b \twoheadrightarrow y_c$ by MVD-transitivity of 2 and 4.
6. $id_x, y_a, \dots, y_b \twoheadrightarrow y_c$ by interaction of 5 and *fd1*.

prefix: The prefix rule states that:

if $R : [x : y_a, \dots, x : y_b, z_l, \dots, z_m \twoheadrightarrow z_c]$ then $R : [x, z_l, \dots, z_m \twoheadrightarrow z_c]$, where $\{y_a, \dots, y_b\} \subseteq \{y_1, \dots, y_k\}$ and $\{z_l, \dots, z_m, z_c\} \subseteq \{z_1, \dots, z_n\}$.

For any relation r of $S(R)$, r satisfies the premise if and only if $flat(r)$ satisfies the following FD:

fd2: $y_a, \dots, y_b, z_l, \dots, z_m \twoheadrightarrow z_c$.

Id_x	y_1	y_k	z_1	z_n
i_1	1	2	b	c
i_2	1	2	d	e

Figure 3.8: Relation that satisfies $Id_x \rightarrow y_1, y_k, Id_x \rightarrow y_1, Id_x \rightarrow y_k$, but not $y_1, y_k \rightarrow Id_x$

We can prove that $id_x, z_l, \dots, z_m \rightarrow z_c$ is a consequence of *mvd1* and *fd2* as follows.

1. $id_x, z_l, \dots, z_m \rightarrow y_1, \dots, y_k, z_l, \dots, z_m$ by MVD-augmentation of *mvd1*.
2. $y_a, \dots, y_b, z_l, \dots, z_m \rightarrow z_c$ by conversion of *fd2*.
3. $y_1, \dots, y_k, z_l, \dots, z_m \rightarrow Y, z_c$, where $Y = \{y_1, \dots, y_k\} - \{y_a, \dots, y_b\}$, by MVD-augmentation of 2.
4. $id_x, z_l, \dots, z_m \rightarrow z_c$ by MVD-transitivity of 1 and 3.
5. $id_x, z_l, \dots, z_m \rightarrow z_c$ by interaction of *fd2* and 4.

singleton: The singleton rule states that:

if $R : [x \rightarrow x : y_1], \dots, R : [x \rightarrow x : y_k]$ then $R : [x : y_1, \dots, x : y_k \rightarrow x]$.

For any relation r of $S(R)$, r satisfies the premise if and only if $flat(r)$ satisfies the following set of FDs:

fd3: $id_x \rightarrow y_i$, for $i \in [1, k]$.

We would have to show that *fd4:* $y_1, \dots, y_k \rightarrow id_x$ is a consequence of *fd3* and *mvd1*. But we can find an instance of $S(R')$, illustrated in Figure 3.8, that satisfies both *fd3* and *mvd1*, but not *fd4*. Therefore, the implication cannot be proved using the inference rules for FDs+MVDs. \square

Intuitively, although expressing levels of nesting using MVDs defined on the flattened relation captures set membership, they do not capture the extensionality property of sets, i.e., that the identity of a set is defined by its elements.

3.4 Discussion

We have presented a definition of functional dependencies (NFD) for the nested relation model. NFDs naturally extend the definition of functional dependencies for the relational model by using path expressions instead of attribute names. The meaning of NFDs was given by defining their translation to logic.

NFDs provide a framework for expressing a natural class of dependencies in complex data structures. Moreover, they can be used to reason about constraints on data integration applications, where both sources and target databases support complex types.

We presented a set of inference rules for NFDs that are sound and complete for the case when no empty sets are present. Although for simplicity we have adopted the nested relational model, and the syntax of NFDs is closely related to this model, allowing nested records or sets would not change the inference rules presented significantly. However, new rules would have to be added to consider path expressions of record types as the current syntax only allows path expressions of set and base types. As an example, we would need a rule that states that if in R , x is a path of type $\langle A_1, \dots, A_n \rangle$, then $R : [x.A_1 \dots x.A_n \rightarrow x]$, where “.” indicates record projection.

Our definition of nested functional dependencies can express both functional dependencies and multi-valued dependencies that capture the hierarchical structure of a nested relation. We believe that the inference rules presented in Section 3.1 are simpler and more intuitive than those for FDs and MVDs considered together. These form of dependencies were adopted for the nested relational model in earlier work ([OY87], [MNE96], [RKS88]) to define normal forms for the nested relational model. In particular, in [MNE96], an entire Section is dedicated to the problem of preventing *singleton buckets* or (singleton sets) in a nested normal form. Since our notion of NFDs can naturally express both FDs, MVDs, and the fact that a set is a singleton, it can therefore form the basis for developing a theory of normal forms for the nested relational model.

In [FSTG85], Fischer, Saxton, Thomas and Van Gucht investigate how nesting defined on a normalized relation destroys or preserves functional and multivalued dependencies; they

also present results on the interaction of inter- and intra-set dependencies. Their results are based on case studies of the cardinality of relations, and of the containment relation between the set of attributes over which the nesting is defined and the set of attributes involved in the dependency. Many results depend on the fact that a nested relation is a singleton set. In our definition of NFDs, both inter- and intra-set dependencies can be expressed. NFDs can also express that a given set is expected to be a singleton. As a result, our work generalizes their results by providing a general framework to reason about interactions between nesting and functional dependencies.

In Section 3.3, we have shown that not all NFDs can be inferred using only inference rules of the relational model for functional and multivalued dependencies. In particular, the *singleton rule* for NFDs, which states that whenever a set is a singleton, the values of its elements determine the value of the set, has no counterpart in the relational model. The previous discussion showed that singleton sets play an important role in the normalization theory and for reasoning about the preservation of FDs under nest and unnest operations. In the context of semi-structured models, such as AceDB[TMD92] and XML[Lay98a], being able to define sets to be singletons is also important. In AceDB [TMD92], a very popular data model among biologists, every label is defined as a set by default. But one can define that a set can contain at most one element. Such a constraint affects the interpretation of update operations. Suppose, for example, that a label A is defined to be a singleton. In this case, an update operation on A to contain value 1 is interpreted as A having a new value of $\{1\}$, no matter what its previous value was. In contrast, if A is not defined to be a singleton, the same operation would have the effect of assigning to A its previous value unioned with $\{1\}$.

In Chapter 6, we will develop a framework to check whether a given FD is guaranteed to hold on a relational schema designed to store XML data, given that certain constraints are valid in the XML document. One of the key parts of the checking algorithm is to determine whether a given element in the document is guaranteed to be unique. For XML documents with no schema information, the constraint language should be able to express that a given element is a singleton. Recently, a number of definitions of FDs for XML has been proposed in the literature [CDHZ03, VLL04, AL04], as well as of MVDs for

XML[ST04]. Some definitions for FDs consider XML documents with a schema, as in [AL04], while others do not assume the existence of one [CDHZ03, LVL03]. But none of them present a complete set of inference rules for their definitions. In particular, [LVL03] presents a set of inference rules and prove that they are complete if FDs are restricted to have a single element on the left-hand side of the FD. In [AL04], it is proved that their definition of FDs is not finitely axiomatizable, and the implication problem is coNP-complete in the general case. Given the difficulties in reasoning about FDs for XML, in the next Chapter we restrict our attention to keys for XML, which is a important special case of FDs.

Chapter 4

Keys for XML

Keys are of fundamental importance in databases. They provide a means for locating a specific object within the database and for referencing one object from another. Keys also constitute an important class of constraints on the validity of data. In particular, keys – as opposed to addresses or internal object identifiers – provide an invariant connection between an object in the real world and its representation in the database. This connection is crucial in modifying the database as the world that it models changes.

As XML is increasingly used in data management, it is natural to require a value-based method of locating an element in an XML document. Key specifications for XML have been proposed in the XML standard [BPSM98], XML Data [Lay98b], and XML Schema [TBMM01]. However, existing proposals cannot handle one or more of the following situations. First, as in databases, one may want to define keys with compound structure. For example, the `name` subelement of a `person` element could be a natural key, but may itself have `first-name` and `last-name` subelements. Keys should not be constrained to be character strings (attribute values). Second, in hierarchically structured data, one may want to identify elements within the scope of a sub-document. For example, the `number` subelement of a `chapter` element may be a key for chapters of a specific book, but would not be unique among chapters of different books. We will call a key such as `number` a *relative key* of `chapter` since it is a key in the context of a given `book` element, i.e., it is

to hold on the sub-document rooted at the `book` element. If the context element is the root of the document, the key is to hold on the entire document, and is referred to as an *absolute key*.

The idea of keys having a scope is not new. In relational databases, scoped keys exist in the form of weak entities. Using the same example, `chapter` is a weak entity of `book`. A chapter number would only make sense in the context of a certain book. Similarly, in the context of the nested relational model, a notion of functional dependencies was introduced in Chapter 2, that can express keys that are local to some levels of nesting. Third, since XML is not required to conform to a DTD or schema definition, it is useful to have a definition of keys that is independent of any specification (such as a DTD or XML Schema) of the type of an XML document.

To overcome these limitations, we propose [BDF⁺02] a new definition of keys for XML which appears to be applicable – among other things – to a wide variety of hierarchical scientific data sets. We understand [Tho02] that these definitions and results have significantly influenced the design of keys in the current version of XML Schema [TBMM01].

In developing this notion of keys for XML, we start with a tree model of data as used in DOM [App98], XSL [Cla99, Wad00], XQL [RLS98] and XML Schema [TBMM01]. An example of this representation is shown in Figure 4.1. Nodes are annotated by their type: *E* for element, *A* for attribute, and *S* for string (or PCDATA). Some keys for this data might include:

1. A `book` node is identified by `@isbn`;
2. An `author` node is identified by `name`, no matter where the `author` node appears;
and
3. Within any subtree rooted at `book`, a `chapter` node is identified by `@number`.

These keys are defined independently of any type specification. The first two are examples of absolute keys since they must hold globally throughout the tree. Observe that `name` has a complex structure. As a consequence, checking whether two authors violate this constraint

```

<db>
  <book isbn=123>
    <title> HTML </title>
    <author> <name>
      <first-name> Tim </first-name>
      <last-name> Bray </last-name>
    </name> </author>
    <chapter number=1> text </chapter>
    .
    .
    <chapter number=10> text </chapter>
  </book>
  <book isbn=234>
    <title> XML </title>
    <author> <name>
      <first-name> Tim </first-name>
      <last-name> Bray </last-name>
    </name> </author>
    <author> <name>
      <first-name> Jean </first-name>
      <last-name> Paoli </last-name>
    </name> </author>
    <chapter number=1> text </chapter>
    .
    .
    <chapter number=12> text </chapter>
  </book>
</db>

```

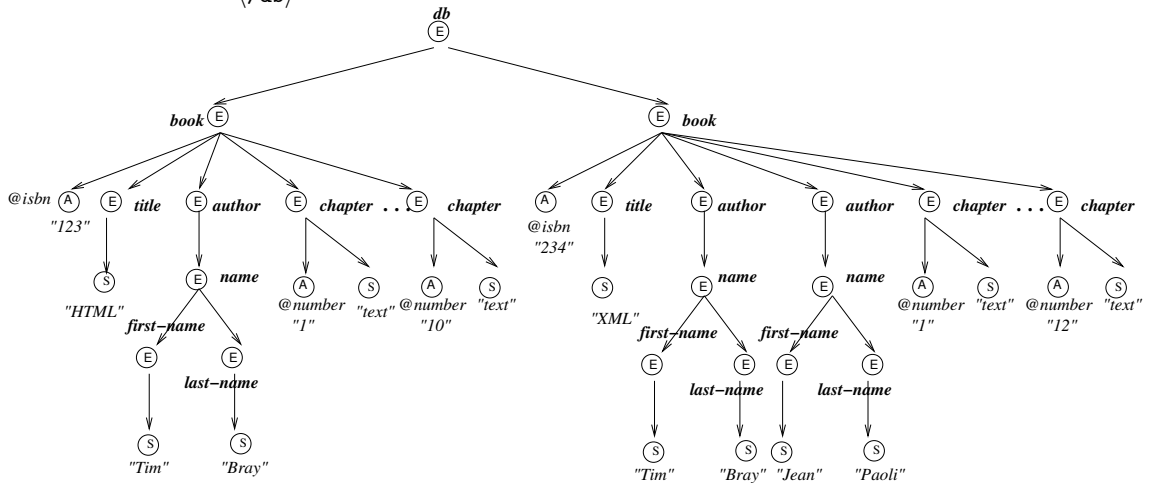


Figure 4.1: Example of some XML data and its representation as a tree

involves testing value-equality on the subtrees rooted at their `name` nodes. The last one is an example of a relative key since it holds locally within each subtree rooted at a `book`. It should be noted that a chapter `@number` is not a key for the set of all `chapter` nodes in the document since two different books have chapters with `@number = 1`. We remark that prior proposals were not capable of expressing the second and third constraints.

The notion of relative keys is particularly natural for hierarchically structured data, and is motivated in part by our experience with scientific data formats. Many scientific databases represent and transmit their data in one of a variety of data formats. Some of these data formats are general purpose, e.g. ASN.1, which is used in GenBank [Ben00] and AceDB [TMD92]; Others, such as EMBL, which is used in SwissProt [Bak00], are domain-specific. All of these specifications have a hierarchical structure. As a typical example, SwissProt [BA00] at the top level consists of a large set of entries, each of which is identified by an accession number. Within each entry there is a sequence of citations, each of which is identified by a number 1,2,3... within the entry. Thus, to identify a citation fully, we need to provide both an accession number for the entry and the number of the citation within the entry. Note that the same number for a citation (e.g. 3) may occur within many different entries, thus the citation number is a relative key within each entry. All the non-XML data formats mentioned above have an easy conversion to XML. We have also found that the data sets in these formats have a natural key structure. However, in many applications XML data does not come with a DTD or schema. This observation supports our claim that, in some applications, keys should be treated independently of any other type constraints.

The remainder of this chapter is organized as follows. Section 4.1 formally defines XML trees, value equality, and (absolute and relative) keys for XML. Section 4.2 describes alternative forms of keys. In Section 4.3 a comparative analysis between keys for XML and Nested Functional Dependencies, discussed in Chapter 2, is presented.

4.1 Keys

Our notion of keys is based on a tree model of XML data, as illustrated in Figure 4.1. Although the model is quite simple, we need to do two things prior to defining keys: The first is to give a precise definition of value equality for XML keys; The second is to describe a path language that will be used to locate sets of nodes in an XML document. We therefore introduce a class of regular path expressions, and define keys in terms of this path language.

4.1.1 A Tree Model and Value Equality

An XML document is typically modeled as a node-labeled tree. We assume three pairwise disjoint sets of labels: \mathbf{E} of element tags, \mathbf{A} of attribute names, and a singleton set $\{\mathbf{S}\}$ denoting text (PCDATA).

Definition 4.1 *An XML tree is defined to be $T = (V, lab, ele, att, val, r)$, where*

- (1) *V is a set of nodes;*
- (2) *lab is a mapping $V \rightarrow \mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$ which assigns a label to each node in V ; a node v in V is called an element (E node) if $lab(v) \in \mathbf{E}$, an attribute (A node) if $lab(v) \in \mathbf{A}$, and a text node (S node) if $lab(v) = \mathbf{S}$;*
- (3) *ele and att are partial mappings that define the edge relation of T . For any node $v \in V$:*
 - *if v is an element then $ele(v)$ is a list of elements and text nodes in V and $att(v)$ is a set of attributes in V ; for each v' in $ele(v)$ or $att(v)$, v' is called a child of v and we say that there is a (directed) edge from v to v' ;*
 - *if v is an attribute or a text node then $ele(v)$ and $att(v)$ are undefined;*
- (4) *val is a partial mapping that assigns a string to each attribute and text node. For any node v in V , if v is an A or S node then $val(v)$ is a string, and $val(v)$ is undefined otherwise;*
- (5) *r is the unique and distinguished root node.*

An XML tree has a tree structure, i.e., for each $v \in V$, there is a unique path of edges from root r to v . An XML tree is said to be finite if V is finite.

For example, Figure 4.1 depicts an XML tree that represents an XML document.

With this, we are ready to define value equality on XML trees. Let $T = (V, lab, ele, att, val, r)$ be an XML tree, and n_1, n_2 be two nodes in V . Informally, n_1, n_2 are value equal if they have the same label and, in addition, either they have the same string value, when they are S or A nodes, or their children are pairwise value equal, when they are E nodes. Formally:

Definition 4.2 *Two nodes n_1 and n_2 are value equal, written $n_1 =_v n_2$, iff the following conditions are satisfied:*

- $lab(n_1) = lab(n_2)$;
- if n_1, n_2 are A or S nodes then $val(n_1) = val(n_2)$;
- if n_1, n_2 are E nodes, then
 - 1) if $att(n_1) = \{a_1, \dots, a_m\}$ then $att(n_2) = \{a'_1, \dots, a'_m\}$ and for all a_i there exists a'_j , $i, j \in [1, m]$, such that $a_i =_v a'_j$; and
 - 2) if $ele(n_1) = [v_1, \dots, v_k]$, then $ele(n_2) = [v'_1, \dots, v'_k]$ and for all $i \in [1, k]$, $v_i =_v v'_i$.
Here, $[v_1, \dots, v_k]$ denotes a list of nodes v_1, \dots, v_k .

That is, $n_1 =_v n_2$ iff their subtrees are isomorphic by an isomorphism that is the identity on string values.

As an example, in Figure 4.1, the `author` subelement of the first `book` and the first `author` subelement of the second `book` are value equal.

4.1.2 Path Languages

There are many options for a path language, ranging from very simple ones, involving just labels, to more expressive ones, such as regular languages or XPath. However, to

develop inference rules for keys, we need to be able to reason about inclusion of path expressions, the so called *containment* problem. It is well known that for regular languages, the containment problem is not finitely axiomatizable [IRS76]; and for XPath, preliminary work [BFK03] has shown that it is not much easier. We therefore restrict our attention to the path language PL , which is expressive enough to capture most practical cases, yet simple enough to be reasoned about efficiently. We will also use a simpler language (PL_s) in defining keys, and therefore show both languages in the table below.

<u>Path Language</u>	<u>Syntax</u>
PL_s	$\rho ::= \epsilon \mid l/\rho$
PL	$q ::= \epsilon \mid l \mid q/q \mid //$

In PL_s , a path is a (possibly empty) list of node labels. Here, ϵ represents the empty path, node label $l \in \mathbf{E} \cup \mathbf{A} \cup \{\mathbf{S}\}$, and “/” is a binary operator that concatenates two path expressions. The language PL_s describes the class of finite lists of node labels. The language PL is a generalization of PL_s that allows the symbol “//”, a combination of wildcard and Kleene closure. This symbol represents any (possibly empty) finite list of node labels. To avoid confusion we write $P//Q$ for the concatenation of P , //, and Q . It should be noted that for any path expression p in any of the path languages, the following equality holds: $p/\epsilon = \epsilon/p = p$. These path languages are fragments of regular expressions [HU79], with PL_s contained in PL .

A path in PL_s or in PL is used to describe a set of paths in an XML tree T . Recall that an attribute node or a text node is a leaf in T and it does not have any child. Thus, a path ρ in PL_s is said to be *valid* if for any label l in ρ , if $l \in \mathbf{A}$ or $l = \mathbf{S}$, then l is the last symbol in ρ . Similarly, we define *valid* path expressions of PL . In what follows, we only consider valid paths and we assume that the regular language defined by a path expression of PL contains only valid paths. For example, $book/author/name$ is a valid path in PL_s and PL , while $//author$ is a valid path expression in PL but it is not in PL_s .

We now give some definitions that will be used throughout the rest of the Chapter.

Definition 4.3 The length of a path ρ in PL_s , denoted by $|\rho|$, is the number of labels in ρ , where the empty path has length 0. By treating “/” as a special label, we also define the length of PL expression P , denoted by $|P|$, to be the number of labels in P .

Definition 4.4 Let ρ be a path in PL_s , and P a path expression in PL . The notation $\rho \in P$ denotes that ρ is in the regular language defined by path expression P .

For example, $book/author/name \in book/author/name$ and $book/author/name \in //name$.

Definition 4.5 Let n_1, n_2 be nodes in an XML tree T . We say that n_2 is reachable from n_1 by following path ρ , denoted by $T \models \rho(n_1, n_2)$, iff (1) $n_1 = n_2$ if $\rho = \epsilon$, and (2) if $\rho = \rho'/l$ then there exists node n in T such that $T \models \rho'(n_1, n)$ and n_2 is a child of n with label l .

We say that node n_2 is reachable from n_1 by following path expression P , denoted by $T \models P(n_1, n_2)$, iff there is a path $\rho \in P$ such that $T \models \rho(n_1, n_2)$.

For example, if T is the XML tree in Figure 4.1, then all the **name** nodes are reachable from the root by following $book/author/name$; They are also reachable by following $//$.

Definition 4.6 Let n be a node in an XML tree T . We use the notation $n[[P]]$ to denote the set of nodes in T that are reachable from n by following path expression P . That is, $n[[P]] = \{n' \mid T \models P(n, n')\}$. We shall use $[[P]]$ as an abbreviation for $r[[P]]$, where r is the root node of T .

For example, let n be the first **book** element in Figure 4.1. Then $n[[chapter]]$ is the set of all **chapter** elements of the first book and $[[//chapter]]$ is the set of all **chapter** elements in the entire document.

Definition 4.7 The value intersection of $n_1[[P]]$ and $n_2[[P]]$, denoted by $n_1[[P]] \cap_v n_2[[P]]$, is defined by:

$$n_1[[P]] \cap_v n_2[[P]] = \{(z, z') \mid z \in n_1[[P]], z' \in n_2[[P]], z =_v z'\}$$

Thus $n_1[[P]] \cap_v n_2[[P]]$ consists of node pairs that are value equal and are reachable by following path expression P starting from n_1 and n_2 , respectively. For example, let n_1 and n_2 be the first and second `book` elements in Figure 4.1, respectively. Then $n_1[[author]] \cap_v n_2[[author]]$ is a set consisting of a single pair (x, y) , where x is the `author` subelement of the first book and y is the first `author` subelement of the second book.

4.1.3 A Key Constraint Language for XML

We are now in a position to define keys for XML and what it means for an XML document to satisfy a key constraint.

Definition 4.8 *A key constraint φ for XML is an expression of the form*

$$(Q, (Q', \{P_1, \dots, P_k\})),$$

where Q , Q' and P_i are PL expressions such that for all $i \in [1, k]$, $Q/Q'/P_i$ is a valid path expression. The path Q is called the context path, Q' is called the target path, and P_1, \dots, P_k are called the key paths of φ .

When $Q = \epsilon$, we call φ an absolute key, and abbreviate the key to $(Q', \{P_1, \dots, P_k\})$; otherwise φ is called a relative key. We use \mathcal{K} to denote the language of keys, and \mathcal{K}_{abs} to denote the set of absolute keys in \mathcal{K} .

As illustrated in Figure 4.2, a key $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$ specifies the following:

- the context path Q , starting from the root of an XML tree T , identifies a set of nodes $[[Q]]$;
- for each node $n \in [[Q]]$, φ defines an absolute key $(Q', \{P_1, \dots, P_k\})$ on the subtree rooted at n ; specifically,
 - the target path Q' identifies a set of nodes $n[[Q']]$ in the subtree, referred to as the *target set*,

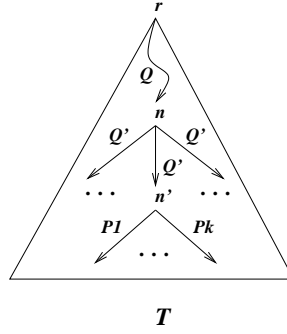


Figure 4.2: Illustration of a key $(Q, (Q', \{P_1, \dots, P_k\}))$

- the key paths P_1, \dots, P_k identify nodes in the target set. That is, for each $n' \in n[[Q']]$ the values of the nodes reached by following the key paths from n' uniquely identify n' in the target set.

For example, the keys on Figure 4.1 mentioned in the beginning of the chapter can be written as follows:

1. `@isbn` is a key of `book` nodes: $(book, \{@isbn\})$;
2. `name` is a key of `author` nodes no matter where they are: $(//author, \{name\})$;
3. within each subtree rooted at a `book`, `@number` is a key of `chapter`:
 $(book, (chapter, \{@number\}))$.

The first two are absolute keys of \mathcal{K}_{abs} and the last one is a relative key of \mathcal{K} .

Definition 4.9 Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$ be a key of \mathcal{K} . An XML tree T satisfies φ , denoted by $T \models \varphi$, iff for any n in $[[Q]]$ and any n_1, n_2 in $n[[Q']]$, if for all $i \in [1, k]$ there exist nodes $x \in n_1[[P_i]]$, $y \in n_2[[P_i]]$ such that $x =_v y$, then $n_1 = n_2$. That is,

$$\forall n \in [[Q]] \quad \forall n_1 n_2 \in n[[Q']] \quad \left(\left(\bigwedge_{1 \leq i \leq k} n_1[[P_i]] \cap_v n_2[[P_i]] \neq \emptyset \right) \rightarrow n_1 = n_2 \right).$$

As mentioned earlier, the key φ defines an absolute key on the subtree rooted at each node n in $[[Q]]$. That is, if two nodes in $n[[Q']]$ are distinct, then the two sets of nodes reached on some P_i must be disjoint (by value equality.) More specifically, for any $n \in [[Q]]$ and for

any distinct nodes n_1, n_2 in $n[[Q']]$, there must exist some P_i , $1 \leq i \leq k$, such that for all x in $n_1[[P_i]]$ and y in $n_2[[P_i]]$, $x \neq_v y$.

Observe that when $Q = \epsilon$, i.e., when φ is an absolute key, the set $[[Q]]$ consists of a unique node, namely, the root of the tree. In this case $T \models \varphi$ iff

$$\forall n_1 n_2 \in [[Q']] \left(\left(\bigwedge_{1 \leq i \leq k} n_1[[P_i]] \cap_v n_2[[P_i]] \neq \emptyset \right) \rightarrow n_1 = n_2 \right).$$

As an example, let us consider the keys defined earlier on the XML tree T in Figure 4.1.

1) $T \models (\text{book}, \{\text{@isbn}\})$ because the `@isbn` attributes of the two `book` nodes in T have different string values. For the same reason, $T \models (\text{book}, \{\text{@isbn}, \text{author}\})$. However, $T \not\models (\text{book}, \{\text{author}\})$ because the two books agree on the values of their first author. Observe that the second `book` node has two `author` subelements, and the key requires that none of these `author` nodes is value equal to the `author` of the first `book`.

2) $T \not\models (//\text{author}, \{\text{name}\})$ because the `author` of the first `book` and the first `author` of the second `book` agree on their names but they are distinct nodes. Note that all `author` nodes are reachable from the root by following `//author`. However, $T \models (\text{book}, (\text{author}, \{\text{name}\}))$ because under each `book` node, the same author does not appear twice.

3) $T \models (\text{book}, (\text{chapter}, \{\text{@number}\}))$ because in the subtree rooted at each `book` node, the `@number` attribute of each `chapter` has a distinct value. However, observe that $T \not\models (\text{book}/\text{chapter}, \{\text{@number}\})$ since both `book` nodes have a `chapter` with `@number = 1` but the two `chapter`'s are distinct.

Several subtleties are worth pointing out. First, observe that each key path can specify a *set* of values. For example, consider again $\psi = (\text{book}, \{\text{@isbn}, \text{author}\})$ on the XML tree T in Figure 4.1, and note that the key path `author` reaches two `author` subelements from the second `book` node. In contrast, this is not allowed in most proposals for XML keys, e.g., XML Schema. The reason that we allow a key path to reach multiple nodes is to cope with the semistructured nature of XML data. Second, the key has no impact on those nodes at which some key path is *missing*. Observe that for any $n \in [[Q]]$ and n_1, n_2 in $n[[Q']]$, if P_i is missing at either n_1 or n_2 then $n_1[[P_i]]$ and $n_2[[P_i]]$ are by definition disjoint.

This is similar to *unique constraints* introduced in XML Schema. In contrast to unique constraints, however, our notion of keys is capable of comparing nodes at which a key path may have multiple values. Third, it should be noted that two notions of equality are used to define keys: value equality ($=_v$) when comparing nodes reached by following key paths, and node identity ($=$) when comparing two nodes in the target set. This is a departure from keys in relational databases, in which only value equality is considered.

4.1.4 Transitive Set of Keys

The purpose of keys is to specify uniquely certain components of a document. Obviously, a relative key such as $(book, (chapter, \{@number\}))$ alone does not uniquely identify a particular chapter in the document. However we believe that if we give a book isbn, and a chapter number, we have specified a chapter. It is this intuition that we need to formalize. First, we will introduce the *immediately precedes* relation, and then define the notion of a transitive set of keys.

Definition 4.10 A key $(Q_1, (Q'_1, S_1))$ immediately precedes $(Q_2, (Q'_2, S_2))$ if $Q_2 = Q_1.Q'_1$. Any relative key immediately precedes itself. The precedes relation is the transitive closure of the immediately precedes relation.

Definition 4.11 A set Σ of relative keys is transitive if for any relative key $(Q_1, (Q'_1, S_1)) \in \Sigma$ there is a key $(\epsilon, (Q'_2, S_2)) \in \Sigma$ which precedes $(Q_1, (Q'_1, S_1))$.

Example 4.1 The following set of keys is transitive:

$$\begin{aligned} &(\epsilon, (book, \{@isbn\})) \\ &(book, (chapter, \{@number\})) \\ &(book/chapter, (section, \{@number\})) \end{aligned}$$

This set is not:

$$\begin{aligned}
&(\epsilon, (\text{book}, \{\text{@isbn}\})) \\
&(\text{book}/\text{chapter}, (\text{section}, \{\text{@number}\}))
\end{aligned}$$

□

Observe that any transitive set of keys must contain an absolute key.

In the next Chapter, we will present a set of inference rules and an algorithm for determining if we can infer some keys in the presence of others. As an example, similar to the relational model, the notion of a “superkey” still holds for XML keys: If $(Q, (Q', S))$ is a key, and $S \subseteq S'$, then so is $(Q, (Q', S'))$. Given the ability to determine whether a key can be inferred from others, next we introduce the notion of a *minimum transitive set of keys*.

Definition 4.12 *Let Σ be a set of XML keys. A transitive set of XML keys $\Theta = \{\kappa_1, \dots, \kappa_n\}$ is minimum with respect to Σ if for all $\kappa_i = (Q, (Q', S))$, there exists no $S' \subset S$ such that $(Q, (Q', S'))$ can be inferred from Σ , and moreover, no κ_i is redundant. A key κ_i is redundant if $\kappa_i = (Q, (Q', S))$, and there exists a κ_j in Θ , $\kappa_j = (Q/Q', (Q'', S'))$, and $(Q, (Q'/Q'', S'))$ can be inferred from Σ .*

Intuitively, in Θ , every key has a minimum set of key paths, and moreover, if the set of attributes S' suffices to identify a node in $n[Q'/Q'']$, where $n \in [Q]$, there is no need to first identify a node $n' \in n[Q']$ using the set of attributes S and then identify $n \in n'[Q'']$ using S' . As an example, the first transitive set of keys in Example 4.1.4 is minimum.

4.2 Discussion

There are other ways of defining keys for XML, both more and less restrictive than what we have described in the previous Section. This section presents a discussion on those alternatives, starting with the proposals in XML-Schema.

4.2.1 XML-Schema

XML-Schema includes a syntax for specifying keys which is related to our definition, but there are some substantive differences, even if we ignore the issue of relative keys. Possibly the most important of these is that the language for path expressions is XPath. XPath is a relatively complex language in which one can not only move down the document tree, but also sideways or upwards, not to mention that predicates can be embedded as well.

Recently, it has been shown [MS04, NS03] that it is rather expensive to determine containment of XPath expressions. These issues are important if we want to reason about keys as we do – for quite practical purposes – in relational databases. Indeed, the containment problem is undecidable in the presence of disjunction, DTDs, and variables [NS03], and it is coNP-complete even for a small fragment of XPath in the absence of DTDs [MS04]. For the (finite) axiomatizability of equivalence of XPath expressions, which is important in studying the (finite) axiomatizability of XML key implication, the analysis is even more intriguing [BFK03]. Thus, not surprisingly, reasoning about keys defined in XML Schema is prohibitively expensive: Even for unary keys, i.e., keys defined in terms of a single subelement, the finite satisfiability problem is NP-hard and the implication problem is coNP-hard [AFL02b]. For the entire class of keys of XML Schema, to the best of our knowledge, both the implication and axiomatizability problems are still open. In contrast, we will show in the next Chapter that our definition of keys can be reasoned about efficiently.

Here is a brief summary of the other salient differences between our definitions and the XML-Schema proposal.

Equality. XML Schema restricts equality to text nodes, while we have adopted a more general form of tree equality.

Definition of the target set. In XML-schema the path expression that defines the target set is taken to start at arbitrary nodes. Recall that in a key $(Q, (Q', S))$ in our definition, the context path Q always starts from the root. But it is straightforward to let Q start from an arbitrary node: one needs simply to substitute $//Q$ for Q .

Definition of key paths. XML-Schema talks about a *list* (not a set) of key paths. While this avoids issues of equivalence of XPath expressions, one can construct keys that are, presumably, equivalent, but have different or anomalous presentations. For example (temporarily using [...] for lists): $(person, [firstname, lastname])$, $(person, [lastname, firstname])$, $(person, [lastname, lastname, firstname])$ impose the same constraint. Given the difficulties in determining the equivalence of XPath expressions, there is no general method of saying whether two such specifications are equivalent.

Relative keys. While there is no direct notion of a relative key in XML-Schema, in certain circumstances one can achieve a related effect. Consider for example:

$$(\epsilon, (book, \{ @isbn \}))$$

$$(book, (chapter, \{ @number \}))$$

In XML-Schema one can specify a key for `chapter` as

$$(book.chapter, [@number, up.@isbn]).$$

Here *up* (this is not XPath syntax) is the XPath instruction to move up one node. Thus part of the key is outside of the value of a `chapter` node. One of the inferences one could make for such a specification is that $(book, [isbn])$ is a key *provided* the nodes in the target set all contain at least one `chapter` child node. Again, it is not clear how to reason generally about such specifications.

4.2.2 Strong Keys

The definition of keys we have described in Section 4.1.3 is quite weak, which we believe is in keeping with the semi-structured nature of XML. However, intuitively it does not mirror the requirements imposed by a key in relational databases, i.e. the *uniqueness* of a key and *equality* of key values. We now explore a definition which captures both these requirements.

In a strong key definition, we require that the keys paths exist and are *unique*. That is, given a key $(Q, (Q', \{P_1, \dots, P_n\}))$ and a node n in $\llbracket Q \rrbracket$, $n \llbracket P_i \rrbracket$ contains exactly one node

for $1 \leq i \leq n$. The key paths constrain the target set as follows: Take any two nodes $(n_1, n_2) \in \llbracket Q \rrbracket$ and consider the pairs of nodes found by following a key path P_i from n_1 and n_2 . If all such pairs of nodes are value-equal, then the nodes n_1 and n_2 are the same node.

As an example of what it means for a path expression to be unique, consider Figure 4.1: at the second `book` element, `title` and `@isbn` are unique, but `author` and `chapter` are not unique at this node.

The definition of satisfaction for strong keys now becomes the following.

Definition 4.13 *Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$ be a key of \mathcal{K} . An XML tree T satisfies φ iff for all nodes n in $\llbracket Q \rrbracket$, n satisfies the following conditions:*

- For all n' in $n\llbracket Q \rrbracket$ and for all $P_i (1 \leq i \leq k)$, P_i is unique at n' .
- For any n_1, n_2 in $n\llbracket Q \rrbracket$, if $n_1\llbracket P_i \rrbracket =_v n_2\llbracket P_i \rrbracket (1 \leq i \leq k)$ then $n_1 = n_2$.

To distinguish the two definitions of keys let us refer to keys defined above as *strong keys* and the keys defined in Section 4.1.3 as *weak keys*. Given this strong notion of keys, let us re-examine some examples given before.

1. $(book, \{\text{@isbn}\})$: any two `book` elements have unique `@isbn` and differ on their values.
2. $(//author, \{name\})$: any two `authors`, no matter where they occur, have unique `name` subelements and differ on those elements.

Now consider the following key: $(//, \{k\})$. It requires that every element have a key k , including any element whose name is k . That is, the key imposes an infinite chain of `k` nodes and therefore, there is no finite document satisfying it. That is, the key is not *finitely satisfiable*. The problem arises because we require that key paths must exist. It should be mentioned that the corresponding key in XML-Schema, $(//*, [id])$, is not meaningful either, because an `id` node cannot have a base type if it is to have an `id` subelement itself.

In order to obtain finite satisfiability of strong keys, we can restrict key paths to be simple attributes, since attributes are always leaves in an XML tree. In this restricted form, keys are of the form

$$(Q, (Q', \{@A_1, \dots, @A_k\}))$$

and we denote this class of keys as \mathcal{K}_{att} .

4.2.3 Keys that Determine Value Equality

So far we have assumed that key equality implies node identity; but occasionally we want key equality to imply value equality. This happens in “non-second-normal form” keys as the following example illustrates. Consider a scientific database that consists of a sequence of entries (each entry describes some structure, e.g., a gene) and within each entry there is a list of citations. The key for citations would be: $(db.entry.citation, \{isbn\})$, and two entries may contain citations with the same *isbn*. Here we do not want to insist that the two citations are the same node, but rather that they are value-equal. Of course, such a database now has redundancy, but allowing occasional redundancies of this kind may be preferable to having a separate list of citations and doing a join in order to recover the citations relevant to an entry. An analog of this happens in relational databases where, for efficiency purposes, it is sometimes useful to have non-second-normal-form relations.

This form of keys is closer to the notion of nested functional dependencies (NFDs) discussed in Chapters 2 and 3, since equality of certain values imply value equality. A comparison analysis between NFDs and XML keys is the subject of the next Section.

4.3 Comparison with Nested Functional Dependencies

Although nested functional dependencies (NFDs) and XML keys are defined on different data models, there are a number of similarities between them. First, both are defined in terms of navigation paths. For NFDs, traversing a path involves going down levels of nestings that are always sets. For XML keys, traversing a path involves going down nested lists, since in XML the order of elements is significant.

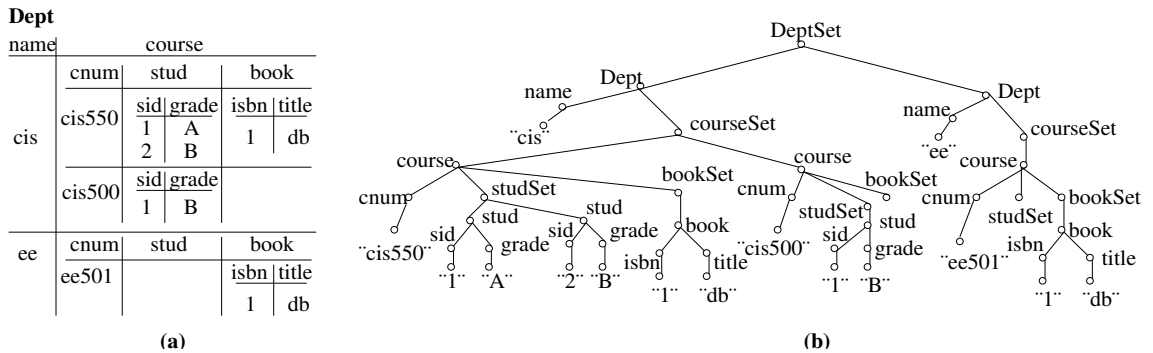


Figure 4.3: Data represented as a nested relation and as an XML tree

Second, both can define constraints defined on different levels of the data. In other words, they can define global as well as local constraints. As an example, consider the data illustrated in Figure 4.3. There is an obvious correspondence between data represented in Figures 4.3(a) and (b), where every set-valued attribute A in the nested relation schema $Dept$ has a corresponding element named $ASet$ in its representation as an XML tree. We can define that $name$ uniquely identify a *department* in the entire database as an NFD: $Dept[name \rightarrow course]$, and as an XML key: $(Dept, \{name\})$. It is possible to express that sid identifies a student within each course as an NFD: $Dept : Course[stud : sid \rightarrow stud : grade]$, and as an XML key: $(Dept/CourseSet/course, (studSet/stud, \{sid\}))$.

Third, both involve equality on complex data. In NFDs, both the right and left-hand side can involve equality of values of set type. XML keys do not restrict keys to be of type text, but may involve equality of trees.

Fourth, both are capable of expressing that a set (or list) must contain at most one element and that the intersection of elements is empty. In the example of Figure 4.3, one can define that each course can have at most one textbook as an NFD: $Dept : Course[book \rightarrow book : isbn]$ (given that $isbn$ is a local key for $book$), and as an XML key: $(Dept/courseSet/course, (bookSet/book, \{\}))$. We can also express that departments do not share course numbers, that is, that intersection of the set of course numbers of different departments is empty. As an NFD this constraint is expressed as: $Dept[course : cnum \rightarrow name]$ (given that $name$ is a key for $Dept$), and as an XML key as: $(Dept, \{courseSet/course/cnum\})$.

```

<!DOCTYPE DeptSet [
  <!ELEMENT DeptSet (Dept*)>
  <!ELEMENT Dept (name, courseSet) >
  <!ELEMENT courseSet (course*)>
  <!ELEMENT course (cnum, studSet, bookSet)>
  <!ELEMENT studSet (stud*)>
  <!ELEMENT stud (sid, grade)>
  <!ELEMENT bookSet (book*)>
  <!ELEMENT book (isbn, title)>
  <!ELEMENT name (#PCDATA)>
  ...
  <!ELEMENT title (#PCDATA)>
]>

```

Figure 4.4: DTD of the document in Figure 4.3

But there are also a number of differences between NFDs and XML keys. The most distinctive characteristic is that NFDs are defined on the nested relational model and require a schema, whereas XML keys are defined on a semi-structured data and does not necessarily come with a schema definition. The nested relational model we have adopted in Chapters 2 and 3 contains both tuple and set constructors, where the outermost level is a set of tuples. In contrast, the XML data model is a node-labeled tree, where internal nodes contain an element label, and leaves can either contain an element label, a string value or an attribute label combined with a string value. But, as illustrated by Figure 4.3, there is an easy translation of a nested relation to an XML document, as well as a translation of a nested relational schema to a DTD. As an example, the XML tree of Figure 4.3 conforms to the DTD in Figure 4.4.

We have shown in Chapter 3 that given a set Σ of NFDs defined on a schema Sc one can always find an instance I of Sc such that I satisfies Σ . That is, NFDs have the *finite satisfiability property*. On the other hand, this is not true for XML keys in the presence of a DTD. To illustrate, let us consider a simple DTD D :

```
<!ELEMENT foo (X, X)>
```

and a simple (absolute) key $\varphi = (X, \emptyset)$. Obviously, there exists a finite XML tree that conforms to the DTD D (see, e.g., Figure 4.5 (a)), and there exists a finite XML tree that satisfies the key φ (e.g., Figure 4.5 (b)). However, there is no XML tree that both

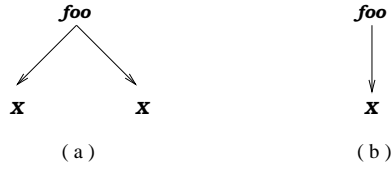


Figure 4.5: Interaction between DTDs and XML keys

conforms to D and satisfies φ . This is because D requires an XML tree to have two distinct X elements, whereas φ imposes the following restriction: The path X , if it exists, must be unique at the root. This shows that in the presence and absence of DTDs, the analysis of key satisfiability and implication can be wildly different. It should be mentioned that keys defined in other proposals for XML, such as those introduced in XML Schema [TBMM01], also interact with DTDs or other type systems for XML. This issue was recently investigated in [AFL02a, FL02] for a class of keys and foreign keys defined in terms of XML attributes. It is shown that in general it is not possible to check whether there exists an XML document that both conforms to the DTD and satisfies a set of keys and foreign keys. In particular, for unary keys and foreign keys the problem is shown to be NP-complete. In contrast, in the next Chapter we will show that the XML keys defined in Section 4.1.3, which are independent of any type definition, are always finite satisfiable, and moreover, there exists a finite set of inference rules that is sound and complete.

The other distinctive characteristic is that although NFDs only involve value-based equality, XML keys involve both value and identity-based equality. As an example, consider again the data illustrated in Figure 4.3. It can be easily verified that the nested relation satisfies the NFD $Dept[course : book : isbn \rightarrow course : book]$. That is, whenever two courses agree on a book's *isbn* then the values of their *book* attribute are equal as sets. The immediate translation of this NFD to an XML key would be $(Dept/courseSet/course/bookSet, \{book/isbn\})$. But this key is not satisfied by the XML tree of Figure 4.3(b) because there are two values *book/isbn* rooted at *Dept/courseSet/course/book* that have the value "1". This is because the value of *book/isbn* determines the *value* of the subtrees rooted at *Dept/courseSet/course/book*, but not their *identity* (as a node tree), which is what the XML key defines. If the XML key definition were modified to require that equal keys imply node equality instead of node identity, as described in Section 4.2.3, then the XML

key above would be satisfied. It would also allow us to express that redundant data should be consistent across the database. As an example, it would be able to express that if two courses agree on their book’s *isbn* then they must also agree on the book’s *title*. This constraint can be defined as an NFD as $Dept[course : book : isbn \rightarrow course : book : title]$, but cannot be expressed in our definition of XML keys. The issue of value equality versus node equality also arises in the definitions for functional dependencies for XML, as we will discuss in the next Section.

4.4 Functional Dependencies for XML

Several definitions of functional dependencies for XML (XFDs) have recently been proposed [CDHZ03, VLL04, AL04]. In [VLL04] and [AL04] XFDs are of the form $X \rightarrow y^1$, where $X \cup \{y\}$ are simple path expressions in PL_s . While XFDs of [AL04] presumes the existence of a DTD, in [VLL04] the definition of the constraint is independent of any type specification. The satisfaction of XFDs are similar to the satisfaction of FDs (and NFDs): given two elements n_1, n_2 in $\llbracket y \rrbracket$, if the values of all $x \in X$ agree, where x is defined along the path from the root to n_1 and n_2 , then $n_1 = n_2$. Since XML documents can have multiple nodes reached by following a path $x \in X$, the “agreement” here is interpreted as *intersection* of sets of values or node identity. Value equality is only applied to leaf nodes with a string value, while node identity is applied in all other nodes. As an example, the document in Figure 4.6 satisfies the the XFD $root.enrollment.lecturer.S \rightarrow root.enrollment$ in both proposals. Here, the label S in the path expression is to obtain the text value associated with element *lecturer*. Observe that value equality is used to compare $root.enrollment.lecturer.S$ and node identity is applied to $root.enrollment$. That is, the interpretation of this XFD is equivalent to the absolute key $(enrollment, \{lecturer\})$. In fact, a subclass of XML keys in which key paths are restricted to paths ending in text values can be captured by XFDs [VLL04]. But the two proposals differ on when an “incomplete” XML tree satisfies an XFD, where

¹In this section we will only consider XFDs with a single path on the right-hand side, although some of the proposals allow multiple paths.

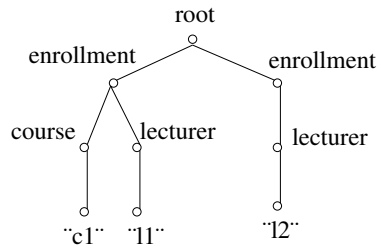


Figure 4.6: Enrollment document

“incomplete” means that elements reached by following an identical path have a different set of subelements and/or attributes. In the definition of [VLL04], an XML tree T satisfies an XFD if for all possible completions of T , the XFD is satisfied. In contrast, for [AL04] an XFD is trivially satisfied if an element on the left-hand side of the XFD is missing. Borrowing the example from [VLL04], the document illustrated in Figure 4.6 satisfies the XFD $root.enrollment.course.S \rightarrow root.enrollment.lecturer.S$ under the semantics defined in [AL04], but not under the semantics of [VLL04], since the completion of the second *enrollment* element with *course* = “c1” would invalidate the XFD.

The syntax of XFDs proposed by [CDHZ03] differs from the definitions previously discussed because an expression $X \rightarrow y$ is associated with variable bindings. The two examples above on the document of Figure 4.6 are expressed as:

$$\begin{aligned}
 &x \in root.enrollment, \$x.lecturer.S \rightarrow \$x, \text{ and} \\
 &x \in root.enrollment, \$x.course.S \rightarrow \$x.lecturer.S.
 \end{aligned}$$

While the path expression bound to a variable can contain “//”, a combination of wildcard and Kleene closure, path expressions in $X \rightarrow y$ are simple paths. In contrast with the previous two definitions, for an XFD to be satisfied, given two nodes n_1, n_2 , reached by following the variable bindings, each of the paths in $X \cup \{y\}$ must exist and be unique. Therefore, although the first XFD is satisfied by the document in Figure 4.6, the second is not, since there exist no *course* subelement under the second *enrollment*. The purpose of XFDs in [AL04] and [VLL04] is to define a normal form for XML documents. In contrast, the purpose of XFDs in [CDHZ03] is to be able to design a relational schema to store XML, reducing the redundancy of data, while enforcing such constraints using relational primary key constraints.

Although none of the definitions discussed in this Section presents a set of complete inference rules, reasoning about them play an important role both in the algorithms for designing the relational schema for XML storage and in the algorithms for normalizing an XML document. In contrast, our definition of keys have a set of inference rules that are sound and complete, as we will discuss in the next Chapter. The ability to reason about keys efficiently will also give us the basis for developing algorithms for computing a set of functional dependencies that are proven to be satisfied by the relational storage of XML data, given that the document satisfies a set of XML keys. This is the subject of Chapter 6.

Chapter 5

Reasoning about Keys for XML

In this chapter we investigate the logical implication and satisfiability of keys for XML. In relational and nested relational databases, the finite satisfiability problem is trivial: Given any finite set of keys, one can always find a finite instance of the schema that satisfies that set. In Chapter 2 we have discussed the implication problem of functional dependencies for the nested relational model. The implication problem for relational keys and, more generally, functional dependencies is also straightforward. It is well-known [AHV95, RG00] that the problem is decidable in linear time, and that there are exactly two inference rules that are sound and complete for the implication analysis. Let R be a relation schema and $Att(R)$ denote the set of attributes of R . We use $X \rightarrow R$ to denote that X is a key of R , where $X \subseteq Att(R)$. Then the rules can be given as:

$$\frac{}{Att(R) \rightarrow R} \quad (\text{schema}) \qquad \frac{X \rightarrow R \quad X \subseteq Y}{Y \rightarrow R} \quad (\text{superkey})$$

The first rule says that for any relation schema R , the set of all the attributes of R is a key of R . The second asserts that if X is a key of R then so is any superset of X .

For XML the story is more complicated since the hierarchical structure of data is far more complex than the 1NF structure of relational data. In some proposals, as in XML Schema, keys are not even finitely satisfiable, as we have discussed in the previous Chapter. The

reason that the implication problem in other proposals is different from the definition of XML keys presented in Chapter 4 is that there are bad interactions between DTDs and keys, as observed in [AFL02a, FL02], and we consider keys independent of any type specification. In this Chapter, we will show that our definition of weak keys defined in Section 4.1.3, and strong keys with only attributes as key paths, defined in Section 4.2.2, can be reasoned about efficiently. More specifically, we show that they are finitely satisfiable and their implication is decidable in PTIME. Better still, their (finite) implication is *finitely axiomatizable*, i.e., there is a finite set of inference rules that is sound and complete for implication of these keys. In developing these results, we also investigate containment of path expressions, which are not only interesting in their own right but also important in the study of decision problems for XML keys.

Despite the importance of reasoning about keys for XML, little previous work has investigated this issue. The only one closely related to this work is [AFL02a, FL02, FS03]. For a class of keys and foreign keys, the decision problems were studied in the absence [FS03] and presence [AFL02a, FL02] of DTDs. The keys considered there are defined in terms of XML attributes and are not as expressive as keys studied in this work.

The remainder of this chapter is organized as follows. The decision problems for key constraints are defined in Section 5.1; Section 5.2 establishes the finite axiomatizability and complexity results for weak keys: First, we give a quadratic time algorithm for determining inclusion of path expressions. The ability to determine inclusion of path expressions is then used in developing inference rules for keys, for which a PTIME algorithm is given. Section 5.2.4 establishes the results for strong keys, followed by discussions in Section 5.3.

5.1 Decision Problems

In this Section we will define the decision problems for our key constraint languages \mathcal{K} , presented in Section 4.1.3, and \mathcal{K}_{att} , presented in Section 4.2.2. We first consider satisfiability of keys of these languages. Let Σ be a finite set of keys in \mathcal{K} (or in \mathcal{K}_{att}), and T be an XML tree. Following [End72], we use $T \models \Sigma$ to denote that T *satisfies* Σ . That is, for

any $\psi \in \Sigma$, $T \models \psi$.

The *satisfiability problem* is to determine, given any finite set Σ of keys in \mathcal{K} (or in \mathcal{K}_{att}) whether there exists an XML tree satisfying Σ . The *finite satisfiability problem* is to determine whether there exists a finite XML tree satisfying Σ .

As observed in the previous section, keys defined in some proposals (e.g., XML Schema) may not be finitely satisfiable. In contrast, any set of key constraints of \mathcal{K} and of \mathcal{K}_{att} can always be satisfied by a finite XML tree. In particular, for any set Σ of keys in \mathcal{K} , a tree consisted of only the root node satisfies Σ . Similarly, a tree T consisted of the root node with its attributes that are required to exist satisfies any set of keys Θ in \mathcal{K}_{att} . That is, in the construction of T , nodes corresponding to attributes $@A_1, \dots, @A_n$ are created under the root, for each key in Θ of the form $(\epsilon, \{@A_1, \dots, @A_n\})$. Such a key determines that attributes $@A_1, \dots, @A_n$ are required to exist under the root node, according to the definition of strong keys satisfaction. Thus,

Observation. Any finite set Σ of keys in \mathcal{K} or in \mathcal{K}_{att} is finitely satisfiable.

Next, we consider implication of key constraints. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys. We use $\Sigma \models \varphi$ to denote Σ *implies* φ , that is, for any XML tree T , if $T \models \Sigma$, then $T \models \varphi$.

There are two implication problems associated with keys: The *implication problem* is to determine, given any finite set of keys $\Sigma \cup \{\varphi\}$, whether $\Sigma \models \varphi$. The *finite implication problem* is to determine whether Σ *finitely implies* φ , that is, whether for any finite XML tree T , if $T \models \Sigma$, then $T \models \varphi$.

Given any finite set $\Sigma \cup \{\varphi\}$ of keys in \mathcal{K} or in \mathcal{K}_{att} , if there is an XML tree T such that $T \models \bigwedge \Sigma \wedge \neg \varphi$, then there must be a finite XML tree T' such that $T' \models \bigwedge \Sigma \wedge \neg \varphi$. That is, key implication has the finite model property and as a result:

Proposition 5.1 *The implication and finite implication problems for keys coincide.*

Proof. First, we will consider keys in \mathcal{K} . Observe that given any finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, $\Sigma \models \varphi$ iff no XML tree T exists such that $T \models \bigwedge \Sigma \wedge \neg \varphi$. Thus it suffices to

show that if there exists an XML tree T such that $T \models \bigwedge \Sigma \wedge \neg\varphi$, then there must be a finite XML tree T' such that $T' \models \bigwedge \Sigma \wedge \neg\varphi$. That is, the complement of the implication problem for \mathcal{K} has the finite model property [End72]. This can be verified as follows. Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. Since $T \not\models \varphi$, there are nodes $n \in \llbracket Q \rrbracket$, $n_1, n_2 \in n \llbracket Q' \rrbracket$, $x_i \in n_1 \llbracket P_i \rrbracket$ and $y_i \in n_2 \llbracket P_i \rrbracket$ for all $i \in [1, k]$ such that $x_i =_v y_i$ but $n_1 \neq n_2$. Let T' be the finite subtree of T that consists solely of all the nodes in the paths from root to x_i, y_i for all $i \in [1, k]$. It is easy to verify that $T' \models \Sigma$ and $T' \models \neg\varphi$. Clearly, T' is a finite XML tree.

The proof for keys in \mathcal{K}_{att} is similar, but in the construction of T' , the finite subtree of T , besides keeping the nodes in the paths from root to x_i and y_i , T' consists also of all required attributes of such nodes. That is, for any $n \in \llbracket Q_1 \rrbracket$, such that there exists Q_2 where x_i or y_i is in $n \llbracket Q_2 \rrbracket$, the following is true: if $Q_1 = Q'_1/Q''_1$ and $\Sigma \models (Q'_1, (Q''_1, \{@A_1, \dots, @A_n\}))$, then given that $T \models \Sigma$, n has as children nodes corresponding to all attributes $@A_1, \dots, @A_n$. Then, in the construction of T' , n will keep as children all the $@A_i$'s nodes. Clearly, T' is a finite XML tree, $T' \models \Sigma$ and $T' \models \neg\varphi$. \square

In light of Proposition 5.1, we can also use $\Sigma \models \varphi$ to denote that Σ finitely implies φ . We investigate the finite implication problems for keys in the next section.

5.2 Key Implication

In this section, we study the finite implication problem for keys. Our main results are the following:

Theorem 5.1 *The finite implication problem for \mathcal{K} is finitely axiomatizable and decidable in polynomial time in the size of keys.*

Theorem 5.2 *The finite implication problem for \mathcal{K}_{att} is finitely axiomatizable and decidable in polynomial time in the size of keys.*

We prove the first theorem by showing that there is a finite axiomatization (see Lemma 5.6) and an algorithm for determining finite implication of \mathcal{K} constraints (see Lemma 5.7). Analogously, we prove the second theorem by showing that there is a finite axiomatization and an algorithm for determining finite implication of \mathcal{K}_{att} constraints (see Lemma 5.8).

A roadmap for the proofs of the theorems is as follows. Since our axioms for finite implication for \mathcal{K} and \mathcal{K}_{att} rely on path containment, we shall first study the containment of path expressions for the language PL in Section 5.2.1. We then provide a finite set of inference rules and show that it is sound and complete for finite implication of \mathcal{K} constraints in Section 5.2.3. Based on the inference rules, we also develop a polynomial time algorithm for determining finite implication. We shall also present complexity results in connection with finite implication of absolute keys of \mathcal{K}_{abs} in Section 5.2.2. We then provide a finite set of inference rules, as well as a polynomial time algorithm for determining implication of \mathcal{K}_{att} in Section 5.2.4. In the remainder of the Chapter we will refer to weak keys of \mathcal{K} as just *keys*, and to keys of \mathcal{K}_{att} as *strong keys*.

5.2.1 Inclusion of PL Expressions

A path expression P of PL is said to be *included* (or *contained*) in another PL expression Q , denoted by $P \subseteq Q$, if for any XML tree T and any node n in T , $n[[P]] \subseteq n[[Q]]$. That is, the nodes reached from n by following P are contained in the set of nodes reached by following Q from n . We write $P = Q$ if $P \subseteq Q$ and $Q \subseteq P$.

In the absence of DTDs, $P \subseteq Q$ is equivalent to the containment of the regular language defined by P in the regular language defined by Q . Indeed, if there exists a path ρ such that $\rho \in P$ but $\rho \notin Q$, then one can construct an XML tree T with a path ρ from the root. It is obvious that in T , $[[P]] \not\subseteq [[Q]]$. The other direction is immediate. Therefore, $P \subseteq Q$ iff for any path ρ , if $\rho \in P$ then $\rho \in Q$.

We investigate inclusion (containment) of path expressions in PL : Given any PL expressions P and Q , is it the case that $P \subseteq Q$? As we shall shortly establish, this is important to the proof of Theorem 5.1, and it is decidable with low complexity.

$\frac{P \in PL}{\epsilon/P \subseteq P \quad P \subseteq \epsilon/P \quad P/\epsilon \subseteq P \quad P \subseteq P/\epsilon}$	(empty-path)
$\frac{P \in PL}{P \subseteq P}$	(reflexivity)
$\frac{P \in PL}{P \subseteq //}$	(star)
$\frac{P \subseteq P' \quad Q \subseteq Q'}{P/Q \subseteq P'/Q'}$	(composition)
$\frac{P \subseteq Q \quad Q \subseteq R}{P \subseteq R}$	(transitivity)

Table 5.1: \mathcal{I}_p : Rules for PL expression inclusion

We provide in Table 5.1 a set of inference rules, denoted by \mathcal{I}_p , and develop a quadratic time algorithm for testing inclusion of PL expressions.

Theorem 5.3 *Given two PL expressions P and Q , \mathcal{I}_p is a set of sound and complete inference rules for determining whether $P \subseteq Q$. Moreover, there is a quadratic time algorithm in the size of P and Q for determining whether $P \subseteq Q$.*

Proof. The soundness of \mathcal{I}_p is easily verified by induction on the lengths of \mathcal{I}_p -proofs. The proof of completeness of \mathcal{I}_p is more involved. Our goal is to establish that if $P \subseteq Q$, then this can be proved by applying rules of \mathcal{I}_p . A roadmap for the proof is as follows: First, nondeterministic finite state automata $M(P)$ and $M(Q)$ are defined for P and Q , respectively. Then, we show that there exists a *simulation relation* (see Definition 5.1) between the start states of $M(P)$ and $M(Q)$ if and only if $P \subseteq Q$ (see Lemma 5.1). Finally, we show that the existence of such relation can be established using the rules of \mathcal{I}_p (see Lemma 5.2), thus completing the proof.

A quadratic time algorithm that determines whether a PL expression P is contained in a PL expression Q is given in Figure 5.2. The correctness and the complexity analysis of the algorithm is shown in Lemma 5.3. □

Next, we develop the lemmas used in establishing Theorem 5.3. To simplify discussion, we

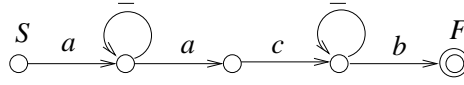


Figure 5.1: NFA for the PL expression $a//a/c//b$

assume that a PL expression P is in *normal form*. A PL expression P is in *normal form* if it does not contain consecutive $//$'s and it does not contain ϵ unless $P = \epsilon$. It is easy to see that given any PL expression P , P can be rewritten into its normal form in linear time as stated by the following proposition.

Proposition 5.2 *Given any PL expression P , it takes linear time to transform P to an equivalent PL expression in normal form.*

Proof. It is easy to see that “ $// //$ ” can be reduced to “ $//$ ” using the *star* and *composition* rules of \mathcal{I}_p . Moreover, by the *empty-path* rule, we can also assume that P does not contain ϵ unless $P = \epsilon$. Obviously, a single pass over P can transform P into its normal form. \square

The proofs of Lemmas 5.1, 5.2, and 5.3 rely on an underlying construction called a *simulation relation*. Given two PL expressions P and Q , a simulation relation is defined on the transition diagrams of the nondeterministic finite state automata (NFAs) [HU79] associated with P and Q . We first describe the NFA associated with a PL expression and then define the simulation.

Let P and Q be path expressions in PL , and the NFAs for P and Q be $M(P)$ and $M(Q)$, respectively, defined as follows:

$$\begin{aligned} M(P) &= (N_1, C \cup \{-\}, \delta_1, S_1, F_1), \\ M(Q) &= (N_2, C \cup \{-\}, \delta_2, S_2, F_2), \end{aligned}$$

where N_1, N_2 are sets of states, C is the alphabet, δ_1, δ_2 are transition functions, S_1, S_2 are start states, and F_1, F_2 are final states of $M(P)$ and $M(Q)$, respectively. Observe that the alphabets of the NFAs have been extended with the special character “ $-$ ” which can match any character in C . Observe also that the transition diagram of a PL expression is always a NFA that has a “linear” structure as depicted in Figure 5.1. More specifically, $M(P)$ has the following properties (similarly for $M(Q)$):

- There is a single final state F_1 .
- For any state $n \in N_1$, except the final state F_1 , there exists exactly one letter $l \in C$ such that the NFA can make a move from n on input l to a single different state n' in N_1 . In other words, $\delta_1(n, l) = \{n'\}$, $n \neq n'$, and $\delta_1(n, l') = \emptyset$ for all $l' \in C$ if $l' \neq l$. For the final state, $\delta_1(F_1, l) = \emptyset$ for all $l \in C$. We shall simply write $\delta_1(n, l) = n'$ if $\delta_1(n, l) = \{n'\}$.
- At any state $n \in N_1$, given the special letter “_”, the NFA either does not move at all, or goes back to n . That is, either $\delta_1(n, _) = \emptyset$ or $\delta_1(n, _) = n$.

As shown in Figure 5.1, the only cycles in the transition diagram of the NFA are introduced by “_”, which go from a state back to itself.

Given $M(P)$ and $M(Q)$, we can define a *simulation relation*, \triangleleft , on $N_1 \times N_2$. Similar to simulations used in the context of semistructured data [ABS00], the relation \triangleleft defines a correspondence between the nodes (or edges) in $M(P)$ and $M(Q)$. Intuitively, the relation \triangleleft is defined in such a way that given an input string, every step taken by $M(P)$ in accepting this string has a corresponding step in $M(Q)$ according to the simulation relation.

Definition 5.1 *Let $M(P)$ and $M(Q)$ be NFAs defined for path expressions P and Q , respectively. Then, for any $n_1 \in N_1$ and $n_2 \in N_2$, there is a simulation relation $n_1 \triangleleft n_2$ if all of the following conditions are satisfied:*

- If $n_1 = F_1$ then $n_2 = F_2$.
- If $\delta_1(n_1, _) = n_1$ then $\delta_2(n_2, _) = n_2$.
- For any $l \in C$, if $\delta_1(n_1, l) = n'_1$ for some $n'_1 \in N_1$, then
 - either there exists a state $n'_2 \in N_2$ such that $\delta_2(n_2, l) = n'_2$ and $n'_1 \triangleleft n'_2$, or
 - $\delta_2(n_2, _) = n_2$ and $n'_1 \triangleleft n_2$.

As Lemma 5.1 will show, given two PL expressions P and Q , showing that $P \subseteq Q$ is equivalent to showing that there is a simulation relation $S_1 \triangleleft S_2$. Observe that by the

definition of the simulation relation, the final state F_1 of $M(P)$ can only correspond to the final state F_2 of $M(Q)$. Therefore, if $S_1 \triangleleft S_2$ and every transition in $M(P)$ corresponds to a transition in $M(Q)$, whenever $M(P)$ accepts an input string, $M(Q)$ also does, and thus $P \subseteq Q$.

Lemma 5.1 *Let P and Q be two PL expressions and $M(P) = \{N_1, C \cup \{-\}, \delta_1, S_1, F_1\}$ and $M(Q) = \{N_2, C \cup \{-\}, \delta_2, S_2, F_2\}$ be their respective NFAs. Then, $P \subseteq Q$ if and only if $S_1 \triangleleft S_2$.*

Proof. The proof makes use of the closure of a transition function δ as defined in [HU79]:

$$\begin{aligned}\hat{\delta}(n, \epsilon) &= \{n\} \\ \hat{\delta}(n, w/l) &= \{p \mid \exists x \in \hat{\delta}(n, w), p \in \delta(x, l)\}\end{aligned}$$

Let $\hat{\delta}_1$ and $\hat{\delta}_2$ be the closure functions of δ_1 and δ_2 , respectively. Observe that $P \subseteq Q$ if and only if for any $\rho \in P$, if $F_1 \in \hat{\delta}_1(S_1, \rho)$ then $F_2 \in \hat{\delta}_2(S_2, \rho)$. Using this observation, we show the lemma as follows. Assume $S_1 \triangleleft S_2$. We first show that if $n_1 \in \hat{\delta}_1(S_1, \rho)$ where ρ is a path in P then there exists $n_2 \in \hat{\delta}_2(S_2, \rho)$ such that $n_1 \triangleleft n_2$. This can be shown by induction on the length of ρ , denoted by $|\rho|$. For the base case, if $\rho = \epsilon$ then by the definition of $\hat{\delta}$, $\hat{\delta}_1(S_1, \epsilon) = \{S_1\}$, and $\hat{\delta}_2(S_2, \epsilon) = \{S_2\}$, and $S_1 \triangleleft S_2$ by assumption. We now assume that the statement is true when $|\rho| < k$ and we shall show that the statement is also true when $|\rho| = k$. Assume $\rho \in P$, $|\rho| > 0$ and let $\rho = \rho'/l$ where $l \in C$. Let $n'_1 \in \hat{\delta}_1(S_1, \rho')$ and by induction hypothesis, there exists $n'_2 \in \hat{\delta}_2(S_2, \rho')$ such that $n'_1 \triangleleft n'_2$. Since $\rho \in P$, ρ is accepted by $M(P)$. Therefore, the last transition taken by $M(P)$ on l from n'_1 to the final state can be one of the following cases:

- l is consumed by a “ $_$ ” transition from n'_1 . More precisely, $\delta_1(n'_1, _) = n'_1$ and by the definition of \triangleleft , it must be that $\delta_2(n'_2, _) = n'_2$. Hence $n'_1 = F_1$ which implies that $n'_2 = F_2$.
- l is consumed by a “ l ” transition from n'_1 . More precisely, $\delta_1(n'_1, l) = F_1$ and by the definition of \triangleleft , either

- for some state $n_2'' \in N_2$, $\delta_2(n_2', l) = n_2''$ and $F_1 \triangleleft n_2''$ which implies that $n_2'' = F_2$
- or
- $\delta_2(n_2', -) = n_2'$ and $F_1 \triangleleft n_2'$ which implies that $n_2' = F_2$.

Thus, if $F_1 \in \hat{\delta}_1(S_1, \rho)$ then we have $F_2 \in \hat{\delta}_2(S_2, \rho)$. That is, $P \subseteq Q$.

For the other direction, we assume $P \subseteq Q$. Our goal is to show that for any path ρ , if $n_1 \in \hat{\delta}_1(S_1, \rho)$ then there exists $n_2 \in \hat{\delta}_2(S_2, \rho)$ such that $n_1 \triangleleft n_2$. To see this, note that for any $\rho \in P$, we have $F_1 \in \hat{\delta}_1(S_1, \rho)$, and since $P \subseteq Q$, $F_2 \in \hat{\delta}_2(S_2, \rho)$. Thus we can define $F_1 \triangleleft F_2$. In addition, for any path ρ , if $\hat{\delta}_1(S_1, \rho) \subseteq N_1$, then there exists path ρ' such that $F_1 \in \hat{\delta}_1(S_1, \rho/\rho')$. Thus the statement can be easily verified by contradiction. Observe that $\hat{\delta}_1(S_1, \epsilon) = \{S_1\}$ and $\hat{\delta}_2(S_2, \epsilon) = \{S_2\}$. Thus $S_1 \triangleleft S_2$ and the lemma follows. \square

The next lemma establishes the relationship between the existence of a simulation relation such that $S_1 \triangleleft S_2$ and the rules of \mathcal{I}_p .

Lemma 5.2 *Let P and Q be two PL expressions and $M(P) = \{N_1, C \cup \{-\}, \delta_1, S_1, F_1\}$ and $M(Q) = \{N_2, C \cup \{-\}, \delta_2, S_2, F_2\}$ be their respective NFAs. If $S_1 \triangleleft S_2$, then $P \subseteq Q$ can be proven using the inferences rules of \mathcal{I}_p .*

Proof. We prove the lemma by first assuming that there exists a simulation relation \triangleleft such that $S_1 \triangleleft S_2$. By the definition of \triangleleft and the properties of $M(P)$, there exists a total mapping $\theta : N_1 \rightarrow N_2$ such that $\theta(S_1) = S_2$, $\theta(F_1) = F_2$, and for any state $n_1 \in N_1$, $n_1 \triangleleft \theta(n_1)$. Let the sequence of states in $M(P)$ be $\vec{v}_1 = p_1, \dots, p_k$, where $p_1 = S_1$ and $p_k = F_1$, and similarly, let the sequence of states in $M(Q)$ be $\vec{v}_2 = q_1, \dots, q_l$, where $q_1 = S_2$ and $q_l = F_2$. It is easy to verify that for any $i, j \in [1, k]$, if $i < j$, $\theta(p_i) = q_{i'}$ and $\theta(p_j) = q_{j'}$, then $i' \leq j'$. We define an equivalence relation \sim on N_1 as follows:

$$p_i \sim p_j \quad \text{iff} \quad \theta(p_i) = \theta(p_j).$$

Let $[p]_{\sim}$ denote the equivalence classes of p with respect to \sim . An equivalence class is *non-trivial* if it contains more than one state. For any equivalence class $[p]$, let p_i and p_j be the smallest and largest states in $[p]$ respectively. That is, for any $p_s \in [p]$, $i \leq s \leq j$.

Algorithm $Incl(n_1, n_2)$

1. if $visited(n_1, n_2)$
then return false
else mark $visited(n_1, n_2)$ as true;
2. process n_1, n_2 as follows:
 - Case (a): if $n_1 = F_1$ then
if $n_2 = F_2$ and $(\delta_1(F_1, -) = \emptyset$ or $\delta_2(F_2, -) = F_2)$
then return true;
else return false;
 - Case (b): if $\delta_1(n_1, a) = n'_1$ and $\delta_2(n_2, a) = n'_2$ for letter a
and $\delta_1(n_1, -) = \emptyset$ and $\delta_2(n_2, -) = \emptyset$
then return $Incl(n'_1, n'_2)$;
 - Case (c): if $\delta_1(n_1, a) = n'_1$ and $\delta_2(n_2, -) = n_2$ and $\delta_2(n_2, a) = n'_2$ for letter a
then return $(Incl(n'_1, n_2)$ or $Incl(n'_1, n'_2))$
else if $\delta_1(n_1, a) = n'_1$ and $\delta_2(n_2, -) = n_2$ and $\delta_2(n_2, a) = \emptyset$
then return $Incl(n'_1, n_2)$;
3. return false

Figure 5.2: Algorithm for testing inclusion of PL expressions

By treating p_i as the start state, and p_j as the final state, we have a NFA that recognizes a regular expression, denoted by $P_{i,j}$. Similarly, we can define $P_{1,i}$ and $P_{j,k}$ such that $P = P_{1,i}/P_{i,j}/P_{j,k}$. It is easy to verify that if $[p]$ is a non-trivial equivalence class, then there must be $\delta_2(\theta(p_i), -) = \theta(p_i)$. In other words, $\theta(p_i)$ indicates an occurrence of “//” in Q . Observe that $P_{1,i}/P_{i,j}/P_{j,k} \subseteq P_{1,i}/P_{j,k}$. This can be proved by using the *star* and *composition* rules of \mathcal{I}_p . By an induction on the number of non-trivial equivalence classes, one can show that $P \subseteq Q$ can always be proved using the *star*, *composition*, *transitivity* and *reflexivity* rules in \mathcal{I}_p as illustrated above. Thus \mathcal{I}_p is complete for inclusion of PL expressions. \square

Based on the previous lemmas, we provide in Figure 5.2 a recursive function $Incl(n_1, n_2)$ for testing inclusion of PL expressions.

Lemma 5.3 *Given two PL expressions P and Q , there exists a quadratic time algorithm for determining whether $P \subseteq Q$.*

Proof. The function $Incl(n_1, n_2)$ is an implementation of Definition 5.1, and assumes the

existence of $M(P')$ and $M(Q')$, where P' and Q' are the normal forms of PL expressions P and Q , respectively. To test whether $P \subseteq Q$, the function $Incl(n_1, n_2)$ is invoked with arguments $Incl(S_1, S_2)$, where S_1 and S_2 are the start states of $M(P')$ and $M(Q')$ respectively. By Lemma 5.1, $P \subseteq Q$ if and only if $S_1 \triangleleft S_2$. Since the function with inputs S_1 and S_2 determines whether $S_1 \triangleleft S_2$, it in fact determines whether $P \subseteq Q$.

We use $visited(n_1, n_2)$ to keep track of whether $Incl(n_1, n_2)$ has been evaluated before. Initially, $visited(n_1, n_2)$ is false for all $n_1 \in N_1$ and $n_2 \in N_2$.

We now show that the algorithm runs in quadratic time. By Proposition 5.2, P and Q can be rewritten into their normal forms in $O(|P|)$ and $O(|Q|)$ time respectively, where $|P|$ and $|Q|$ are the lengths of P and Q . The construction of $M(P)$ can also be done in $O(|P|)$ time and the same argument applies for Q . The initialization statement can be executed in $O(|P| |Q|)$ time. Since each condition of the cases (a)-(c) can be tested in constant time and the first statement of the algorithm ensures that any pair of states (n_1, n_2) from $N_1 \times N_2$ is never processed twice, it is easy to see that $Incl(S_1, S_2)$ runs in $O(|P| |Q|)$ time. We can therefore conclude that the algorithm is in quadratic time. \square

5.2.2 Axiomatization for Absolute Key Implication

Recall that an absolute key (Q', S) is a special case of a \mathcal{K} constraint $(Q, (Q', S))$ when $Q = \epsilon$. Absolute keys are constraints imposed on the entire XML tree T rather than on certain subtrees of T . Not surprisingly, the problem of determining (finite) implication of absolute keys is simpler than that for relative keys. We therefore start by giving a discussion on the rules for absolute key implication. The set of rules, denoted as \mathcal{I}_{abs} , is shown in Table 5.2 and is subsequently extended as rules for relative key implication.

- *superkey*. If S is a key for the set of nodes in $\llbracket Q \rrbracket$ then so is any superset of S . This is the only rule of \mathcal{I}_{abs} that has a counterpart in relational key inference.
- *subnodes*. Observe that any node $v \in \llbracket Q/Q' \rrbracket$ must be in the subtree rooted at some node v' in $\llbracket Q \rrbracket$ and since we have a tree model, there is no sharing of nodes. Hence v

$\frac{(Q, S) \quad P \in PL}{(Q, S \cup \{P\})}$	(superkey)
$\frac{(Q/Q', \{P\})}{(Q, \{Q'/P\})}$	(subnodes)
$\frac{(Q, S \cup \{P_i, P_j\}) \quad P_i \subseteq P_j}{(Q, S \cup \{P_i\})}$	(containment-reduce)
$\frac{(Q, S) \quad Q' \subseteq Q}{(Q', S)}$	(target-path-containment)
$\frac{(Q, S \cup \{\epsilon, P\}) \quad P' \in PL}{(Q, S \cup \{\epsilon, P/P'\})}$	(prefix-epsilon)
$\frac{(Q, \{\})}{(Q/@l, \{\})}$	(attribute)
$\frac{}{(\epsilon, \{\})}$	(epsilon)

Table 5.2: \mathcal{I}_{abs} : Rules for absolute key implication

uniquely identifies v' . Therefore, if a key path P uniquely identifies a node in $\llbracket Q/Q' \rrbracket$ then Q'/P uniquely identifies a node in $\llbracket Q \rrbracket$.

- *containment-reduce.* If $S \cup \{P_i, P_j\}$ is the set of key paths that uniquely identifies nodes in $\llbracket Q \rrbracket$ and $P_i \subseteq P_j$ then we can leave out P_j from the set of key paths. This is because for any nodes n_1, n_2 in $\llbracket Q \rrbracket$, if $n_1 \llbracket P_i \rrbracket \cap_v n_2 \llbracket P_i \rrbracket \neq \emptyset$, then we must have $n_1 \llbracket P_j \rrbracket \cap_v n_2 \llbracket P_j \rrbracket \neq \emptyset$ since $P_i \subseteq P_j$. Thus, by the definition of keys, $S \cup \{P_i\}$ is also a key for $\llbracket Q \rrbracket$.
- *target-path-containment.* A key for the set $\llbracket Q \rrbracket$ is also a key for any subset of $\llbracket Q \rrbracket$. Observe that $\llbracket Q' \rrbracket \subseteq \llbracket Q \rrbracket$ if $Q' \subseteq Q$.
- *prefix-epsilon.* If a set $S \cup \{\epsilon, P\}$ is a key of $\llbracket Q \rrbracket$, then we can extend the key path P by appending to it another path P' , and the modified set is also a key of $\llbracket Q \rrbracket$. This is because for any nodes $n_1, n_2 \in \llbracket Q \rrbracket$, if $n_1 \llbracket P/P' \rrbracket \cap_v n_2 \llbracket P/P' \rrbracket \neq \emptyset$ and $n_1 =_v n_2$, then we have $n_1 \llbracket P \rrbracket \cap_v n_2 \llbracket P \rrbracket \neq \emptyset$. Note that $n_1 =_v n_2$ if $n_1 \llbracket \epsilon \rrbracket \cap_v n_2 \llbracket \epsilon \rrbracket \neq \emptyset$. Thus, by the definition of keys, $S \cup \{\epsilon, P/P'\}$ is also a key for $\llbracket Q \rrbracket$. Observe, however, that the implication of $(Q, \{\epsilon\})$ from the premise is not sound. One can construct an XML tree with only two nodes n_1 and n_2 in $\llbracket Q \rrbracket$ that are value equal but do not have

Algorithm implication

Input: a finite set $\Sigma \cup \{\varphi\}$ of absolute keys, where $\varphi = (Q, \{P_1, \dots, P_k\})$

Output: true iff $\Sigma \models \varphi$

1. if $\varphi = (Q/@l, \{\})$ then $\varphi := (Q, \{\})$;
2. if $Q = \epsilon$ then output true and terminate
3. for each $(Q_i, S_i) \in (\Sigma \cup \{\varphi\})$ do
 - repeat until no further change
 - if $S_i = S \cup \{P', P''\}$ such that $P' \subseteq P''$ then $S_i := S_i \setminus \{P''\}$
4. for each $\phi \in \Sigma$ do
 - (i) if $\phi = (Q', \{P'_1, \dots, P'_m\})$, $Q \subseteq Q'$ and for all $i \in [1..m]$ there exists $j \in [1..k]$ such that either
 - (a) $P_j \subseteq P'_i$ or
 - (b) $P_j = R_1/R_2$, $R_1 \subseteq P'_i$ and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 - (ii) if $\phi = (Q'/Q'', \{P\})$, $Q \subseteq Q'$ and for some $j \in [1..k]$, either
 - (a) $P_j \subseteq Q''/P$ or
 - (b) $P_j = R_1/R_2$, $R_1 \subseteq Q''/P$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 - (iii) if $\phi = (Q'/Q'', \{\})$, $Q \subseteq Q'$ and for some $j \in [1..k]$, either
 - (a) $P_j \subseteq Q''$ or
 - (b) $P_j = R_1/R_2$, $R_1 \subseteq Q''$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
5. output false

Figure 5.3: Finite implication of absolute keys

any paths in P . Since paths of P are missing in the trees of n_1 and n_2 , the XML tree satisfies the premise trivially. However, this tree clearly does not satisfy $(Q, \{\epsilon\})$ since $n_1 =_v n_2$.

- *attribute.* If there exists a single node reached by following path Q , and any node has at most one attribute labeled $@l$, then it is also true that there exists a single node reached by following path $Q/@l$.
- *epsilon.* This rule is sound because there is only one root. In other words, $[\epsilon]$ is exactly the root node and therefore the empty set forms a key for the root.

Observe that these rules are far more complex than the rules for relational key inference (given in the beginning of the Chapter). Moreover, observe that some rules rely on the ability to reason about path inclusion.

As our next theorem shall show, the set of inference rules \mathcal{I}_{abs} is sound and complete for determining the (finite) implication of absolute keys. Moreover, there is an $O(n^5)$ algorithm for determining the (finite) implication of absolute keys, where n is the size of keys.

Theorem 5.4 *The finite implication problem for \mathcal{K}_{abs} is finitely axiomatizable and decidable in $O(n^5)$ time, where n is the size of keys.*

Proof. We omit the proof of soundness and completeness of \mathcal{I}_{abs} because most of the proof can be verified along the same lines as the proof of Lemma 5.6 that we shall discuss in Section 5.2.3.

A function for determining finite implication of absolute keys is given in Figure 5.3. The correctness of the algorithm follows from the axioms for finite implication of absolute keys. Step 1 and 2 of the algorithm are simple applications of the *attribute* and *epsilon* rules, respectively. Step 3 applies *containment-reduce* to transform keys to the key normal form. A key $\phi = (Q, (Q', S))$ of \mathcal{K} is in the *key normal form* if for every pair of paths P_i and P_j in S , $P_i \not\subseteq P_j$. In Step 4, the algorithm checks whether a key ϕ in Σ can prove φ by verifying the applicability of rules of \mathcal{I}_{abs} in three cases: when ϕ has many key paths (Step 4(i)), when ϕ has only one key path (Step 4(ii)), and when ϕ has no key paths (Step 4(iii)). Note that Step 4(i) and Step 4(ii) coincide when $\phi = (Q', \{P'_1\})$.

In Step 4(i), we apply *target-path-containment* rule to infer $(Q, \{P'_1, \dots, P'_m\})$ from ϕ , if possible. If successful, our remaining goal is to transform the set of key paths $\{P'_1, \dots, P'_m\}$ of ϕ to $\{P_1, \dots, P_k\}$ of φ . Since Step 3 has been applied, the set $\{P'_1, \dots, P'_m\}$ cannot be reduced further. Furthermore, observe that at this point, only *superkey*, *containment-reduce*, and *prefix-epsilon* rules are relevant rules for key paths. Our goal is thus to transform every key path P'_i into some P_j using these rules. The resulting set of key paths can be augmented, through the use of *superkey* rule, so that the final set of key paths is $\{P_1, \dots, P_k\}$, as desired. Obviously, a key path P'_i can be replaced with P_j for some $j \in [1, k]$ if $P_j \subseteq P'_i$ (this corresponds to Step 4(i)(a)). The replacement can be done through the use of *superkey* rule, to add the path P_j , and then *containment-reduce*, to remove the path P'_i . Otherwise,

if a proper prefix of P_j is contained in P'_i (see Step 4(i)(b)), then the replacement can occur through the use of *superkey* rule to add the key path R_1 . Then through *containment-reduce*, we remove the path P'_i . The path R_1 in the current set of key paths can then be extended to R_1/R_2 : We first add another key path ϵ (if it does not already exist) through *superkey* rule. Then, through *prefix-epsilon* rule, R_1/R_2 can be obtained. This also explains the requirement that the key path set of φ must contain ϵ . Observe that these are the only possible ways to obtain the desired set of key paths. No inference rules can be applied if P'_i is contained in P_j or a proper prefix of P_j . Thus Step 3(i) is correct in the case when there are many key paths in ϕ .

In Step 4(ii), the *subnode* rule is applied to first obtain $(Q', \{Q''/P\})$ from ϕ and the rest of the argument is similar to the preceding discussion. In Step 4(iii), the *superkey* rule must be applied to obtain $(Q'/Q'', \{\epsilon\})$ before the *subnode* rule can be applied.

Observe that in Step 4, we test whether φ can be proven from Σ by going through each $\phi \in \Sigma$ at most once. This is sufficient because if indeed $\Sigma \models \varphi$, then φ must be the consequence of a sequence of applications of rules of \mathcal{I}_{abs} on a single key in Σ , as illustrated in the previous discussion. That is, since the applicability of *epsilon* and *attribute* rules have already been checked in Steps 1 and 2 and every other rule of \mathcal{I}_{abs} has a premise that consists of only one key, the first rule applied in the sequence of rule applications must have a premise that makes use of only one of the keys in Σ .

We next show that the algorithm runs in $O(n^5)$ where n is the size of keys. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K}_{abs} . Without loss of generality, we assume that all path expressions in the set are in the normal form. If not, by Proposition 5.2, it takes linear time to transform a PL expression to an equivalent PL expression in the normal form. It only takes constant time to execute Steps 1 and 2 of the algorithm. From Theorem 5.3, Step 3 can be done in cubic time. To see this, consider a key $\phi = (Q_i, S_i)$ in $\Sigma \cup \{\varphi\}$. It takes $|S_i| * |S_i|$ units of time to check containment of path expressions in S_i . Since there are at most $|\Sigma| + |\varphi|$ keys, Step 3 is $O(n^3)$ in the size of Σ and φ . Case 4(i) of the algorithm requires one to test for containment of path expressions P_j in P'_i , which can be done in $O(|P_j| * |P'_i|)$ time, and, in case (b), partition P_j in $|P_j|$ possible ways and test for containment in P'_i . This

$\frac{(Q, (Q', S)) \quad P \in PL}{(Q, (Q', S \cup \{P\}))}$	(superkey)
$\frac{(Q, (Q'/Q'', \{P\}))}{(Q, (Q', \{Q''/P\}))}$	(subnodes)
$\frac{(Q, (Q', S \cup \{P_i, P_j\})) \quad P_i \subseteq P_j}{(Q, (Q', S \cup \{P_i\}))}$	(containment-reduce)
$\frac{(Q, (Q', S)) \quad Q_1 \subseteq Q}{(Q_1, (Q', S))}$	(context-path- containment)
$\frac{(Q, (Q', S)) \quad Q_2 \subseteq Q'}{(Q, (Q_2, S))}$	(target-path- containment)
$\frac{(Q, (Q_1/Q_2, S))}{(Q/Q_1, (Q_2, S))}$	(target-to- context)
$\frac{(Q, (Q', S \cup \{\epsilon, P\})) \quad P' \in PL}{(Q, (Q', S \cup \{\epsilon, P/P'\}))}$	(prefix- epsilon)
$\frac{(Q_1, (Q_2, \{Q'/P_1, \dots, Q'/P_k\})) \quad (Q_1/Q_2, (Q', \{P_1, \dots, P_k\}))}{(Q_1, (Q_2/Q', \{P_1, \dots, P_k\}))}$	(interaction)
$\frac{(Q, (Q', \{\}))}{(Q, (Q'/@l, \{\}))}$	(attribute)
$\frac{Q \in PL, S \text{ is a set of } PL \text{ expressions}}{(Q, (\epsilon, \{\}))}$	(epsilon)

Table 5.3: \mathcal{I} : Rules for key implication

requires $O(|P_j| * |P_j| * |P'_i|)$ time. Therefore, for a key $\phi \in \Sigma$, the cost of Case 4(i) is at most $(|P_1| + \dots + |P_k|)(|P'_1| + \dots + |P'_m|) + (|P_1|^2 + \dots + |P_k|^2)(|P'_1| + \dots + |P'_m|)$, which is $O(n^3)$. The cost of Cases 4(ii) and 4(iii) of the algorithm is $O(n^4)$ because they require one to execute the same containment test as Case 4(i) for $|Q'/Q''|$ possible ways to partition Q'/Q'' . Since each constraint ϕ in Σ is examined at most once, the algorithm is $O(n^5)$, where n is the size of Σ and φ . It is possible that this algorithm can be improved further to achieve a lower complexity but this is beyond the scope of this work. \square

5.2.3 Axiomatization for Key Implication

We now turn to the finite implication problem for \mathcal{K} , and start by giving in Table 5.3 a set of inference rules, denoted by \mathcal{I} . Most rules are generalizations of rules shown in Table 5.2 except for rules that deal with the context path in the setting of relative keys: context-path-containment, target-to-context, and interaction. We briefly illustrate these rules below.

- *context-path-containment*. Note that $\llbracket Q_1 \rrbracket \subseteq \llbracket Q \rrbracket$ if $Q_1 \subseteq Q$. If (Q', S) holds on all subtrees rooted at nodes in $\llbracket Q \rrbracket$, then it must also hold on all subtrees rooted at nodes in any subset of $\llbracket Q \rrbracket$.
- *target-to-context*. If a set S of key paths can uniquely identify nodes of a set X in the entire tree T , then it can also identify nodes of X in any subtree of T . Along the same lines, if in a tree T rooted at a node n in $\llbracket Q \rrbracket$, S is a key for $n\llbracket Q_1/Q_2 \rrbracket$, then in any subtree of T rooted at n' in $n\llbracket Q_1 \rrbracket$, S is a key for $n'\llbracket Q_2 \rrbracket$. Note that $n'\llbracket Q_2 \rrbracket$ consists of nodes that are in both $n\llbracket Q_1/Q_2 \rrbracket$ and the subtree rooted at n' . In particular, when $Q = \epsilon$ this rule says that if $(Q_1/Q_2, S)$ holds then so does $(Q_1, (Q_2, S))$. That is, if the (absolute) key holds on the entire document, then it must also hold on any sub-document.
- *interaction*. This is the only rule of \mathcal{I} that has more than one key in its precondition. By the first key in the precondition, in each subtree rooted at a node n in $\llbracket Q_1 \rrbracket$, $Q'/P_1, \dots, Q'/P_k$ uniquely identify a node in $n\llbracket Q_2 \rrbracket$. The second key in the precondition prevents the existence of more than one Q' node under Q_2 that coincide in their P_1, \dots, P_k nodes. Therefore, P_1, \dots, P_k uniquely identify a node in $n\llbracket Q_2/Q' \rrbracket$ in each subtree rooted at n in $\llbracket Q_1 \rrbracket$. More formally, for any $n \in \llbracket Q_1 \rrbracket$ and $n_1, n_2 \in n\llbracket Q_2/Q' \rrbracket$, there must be v_1, v_2 in $n\llbracket Q_2 \rrbracket$ such that $n_1 \in v_1\llbracket Q' \rrbracket$, $n_2 \in v_2\llbracket Q' \rrbracket$ and for all $i \in [1, k]$, we must have $n_1\llbracket P_i \rrbracket \subseteq v_1\llbracket Q'/P_i \rrbracket$ and $n_2\llbracket P_i \rrbracket \subseteq v_2\llbracket Q'/P_i \rrbracket$. If $n_1\llbracket P_i \rrbracket \cap_v n_2\llbracket P_i \rrbracket \neq \emptyset$, then $v_1\llbracket Q'/P_i \rrbracket \cap_v v_2\llbracket Q'/P_i \rrbracket \neq \emptyset$, for any $i \in [1, k]$. Thus, by the first key in the precondition, $v_1 = v_2$. Hence $n_1, n_2 \in v_1\llbracket Q' \rrbracket$ and as a result, $n_1 = n_2$ by the second key in the precondition. Therefore, $(Q_1, (Q_2/Q', \{P_1, \dots, P_k\}))$ holds.

Given a finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, we use $\Sigma \vdash_{\mathcal{I}} \varphi$ to denote that φ is provable from Σ using \mathcal{I} (and \mathcal{I}_p for path inclusion).

To illustrate how \mathcal{I} is used in an implication proof, let us consider two \mathcal{K} constraints:

$$\begin{aligned}\phi &= (A, (B/C//, \{D, D//\})), \\ \psi &= (A/B, (C, \{//D, E\})).\end{aligned}$$

An \mathcal{I} -proof for $\phi \models \psi$ is given as follows.

- 1) $\phi \models (A, (B/C//, \{D\}))$ by $D \subseteq D//$ and the *containment-reduce* rule. Note that $D \subseteq D//$ is proved by using *star*, *empty-path* and *composition* of \mathcal{I}_p .
- 2) $\phi \models (A, (B/C, \{//D\}))$ by 1) and *subnodes*.
- 3) $\phi \models (A/B, (C, \{//D\}))$ by 2) and *target-to-context*.
- 4) $\phi \models (A/B, (C, \{//D, E\}))$ by 3) and *superkey*.

As another example, observe that the following is provable from \mathcal{I} :

$$\frac{(Q, (Q', S \cup \{P\})) \quad P' \subseteq P}{(Q, (Q', S \cup \{P'\}))} \quad (\text{key-path-containment})$$

Indeed, if $(Q, (Q', S \cup \{P\}))$ holds then by *superkey*, so does $(Q, (Q', S \cup \{P, P'\}))$. By *containment-reduce* we have that $(Q, (Q', S \cup \{P'\}))$ holds.

We now show that \mathcal{I} is indeed a finite axiomatization for \mathcal{K} constraint implication. The proof for soundness and completeness is given in Lemma 5.6 and relies on the notion of abstract trees. An *abstract tree* is an extension of an XML tree by allowing “//” as a node label. Abstract trees have the following property, given by Lemma 5.5: whenever a finite abstract tree can be constructed such that it satisfies a set of keys Σ but not a key φ , then an XML tree can be derived with the same property – it satisfies Σ but not φ . Thus this XML tree is a proof witnessing $\Sigma \not\models \varphi$. Given this, to prove that \mathcal{I} is complete for determining (finite) implication of keys, it suffices to show that whenever $\Sigma \not\vdash_{\mathcal{I}} \varphi$, there exists an abstract tree T such that T satisfies Σ but not φ .

We start by giving a discussion on abstract trees. In an abstract tree, “//” is treated as an ordinary label. Therefore, the sequence of labels in an abstract tree is a *PL* expression that may contain occurrences of “//”. Let R be the sequence of labels in the path from

node a to b in an abstract tree T , and let P be a path expression in PL . We say that $T \models P(a, b)$ if $R \subseteq P$. Given this, the definitions of node sets can be easily generalized for abstract trees. Given a node n in an abstract tree T and a PL expression P , the *node set* $n[[P]]$ in T consists of all nodes x such that $T \models P(n, x)$. The satisfaction of a \mathcal{K} constraint for abstract trees uses this definition of node set and is very similar to Definition 4.9. An abstract tree T *satisfies* a key $(Q, (Q', \{P_1, \dots, P_k\}))$ if for every node $n \in [[Q]]$, n satisfies the key $(Q', \{P_1, \dots, P_k\})$. A node n *satisfies* a key $(Q', \{P_1, \dots, P_k\})$ if for any n_1, n_2 in $n[[Q']]$, if for all $i \in [1, k]$ there exist nodes x_i and y_i in T such that $T \models P_i(n_1, x_i)$, $T \models P_i(n_2, y_i)$, and $x_i =_v y_i$, then $n_1 = n_2$.

The following definition describes the construction of an XML tree from an abstract tree.

Definition 5.2 *Given an abstract tree T , and an element tag η , we say that G is the XML tree defined from T using η if G is obtained by substituting every occurrence of “//” in T by η .*

Observe that G and T have the same set of nodes. In addition, for any nodes a, b in G , there is a path ρ such that $G \models \rho(a, b)$ iff there is a path R in T such that $T \models R(a, b)$, where R is the same as ρ except that for each occurrence of “//” in R , the label η appears at the corresponding position in ρ . Let us refer to R as the path expression w.r.t. ρ and conversely, ρ as the path w.r.t. R .

Our goal is to show that, given a set of keys $\Sigma \cup \{\varphi\}$, if T satisfies Σ but not φ , then the XML tree G defined from T using some label η also satisfies Σ and not φ . To do so, we first establish that there is a correspondence between nodes in T and in G reached by following a path expression P in PL . This result is then used to prove the desired property.

Lemma 5.4 *Let T be an abstract tree, η be an element tag that does not occur anywhere in T , and G be the XML tree defined from T using η . Let P be a path expression in PL , and a, b be nodes in G . Then, there exists a path $\rho \in P$ such that $G \models \rho(a, b)$ if and only if $T \models P(a, b)$, i.e., $T \models R(a, b)$ and $R \subseteq P$ where R is the path expression w.r.t. ρ .*

Proof. (1) Assume that $T \models P(a, b)$, i.e., there is a path R from a to b in T such that $R \subseteq P$. By the definition of G , we must have $G \models \rho(a, b)$, where ρ is the path w.r.t. R . Recall that ρ is obtained by substituting η for occurrences of “//”. Since $R \subseteq P$, we have $\rho \in P$.

(2) Conversely, assume that there exists a path $\rho \in P$ such that $G \models \rho(a, b)$. By the definition of G , we have $T \models R(a, b)$, where R is the path expression w.r.t. ρ . Thus, it suffices to show that $R \subseteq P$. To do so, we consider the NFAs of R , P and ρ as defined in Section 5.2.1:

$$\begin{aligned} M(R) &= (N_R, A \cup \{-\}, \delta_R, S_R, F_R), \\ M(P) &= (N_P, A \cup \{-\}, \delta_P, S_P, F_P), \\ M(\rho) &= (N_\rho, A \cup \{\eta\}, \delta_\rho, S_\rho, F_\rho), \end{aligned}$$

where A is an alphabet that contains neither “-” nor η . Recall that NFAs for PL expressions have a “linear” structure as shown in Figure 5.1. In particular, since ρ does not contain “//”, $M(\rho)$ has a strict linear structure. More specifically, let the sequence of states in N_ρ be s_1, \dots, s_m , where $s_1 = S_\rho$ and $s_m = F_\rho$. Then for any $i \in [1, m-1]$, there is exactly one $l \in A \cup \{\eta\}$ such that $\delta_\rho(s_i, l) \neq \emptyset$. More precisely, $\delta_\rho(s_i, l) = s_{i+1}$, and for any $l \in A \cup \{\eta\}$, $\delta_\rho(F_\rho, l) = \emptyset$. Let the sequence of states in N_R be n_1, \dots, n_k , where $n_1 = S_R$ and $n_k = F_R$. Then we can define a function f from N_ρ to N_R with the following properties:

- $f(S_\rho) = S_R$ and $f(F_\rho) = F_R$.
- For any $i, j \in [1, m]$, if $f(s_i) = n_{i'}$, $f(s_j) = n_{j'}$ and $i < j$, then $i' \leq j'$.
- For any $i \in [1, m]$ and $l \in A$, $\delta_\rho(s_i, l) = s_{i+1}$ iff $\delta_R(f(s_i), l) = f(s_{i+1})$ and $f(s_i) \neq f(s_{i+1})$.
- For the special letters “-” and “ η ”, for any $i \in [1, m]$, we let $\delta_\rho(s_i, \eta) = s_{i+1}$ iff $\delta_R(f(s_i), -) = f(s_{i+1})$ and $f(s_i) = f(s_{i+1})$. In particular, if it is the case that $\delta_R(F_R, -) = F_R$ then we have $\delta_\rho(s_{m-1}, \eta) = F_\rho$ and $f(s_{m-1}) = f(F_\rho) = F_R$.

We define an equivalence relation \sim on N_ρ such that

$$s \sim s' \quad \text{iff} \quad f(s) = f(s').$$

Let us use $[s]$ to denote the equivalence class of s w.r.t. \sim . Without loss of generality, assume that R is in the normal form, i.e., it does not contain two consecutive $//$'s and it does not contain ϵ unless it is ϵ . Then it is easy to verify that $[s]$ consists of at most two states. More precisely, if $[s] = \{s\}$, then either s is a final state or there is $l \in A$ such that $\delta_\rho(s, l) = s'$, and if $[s] = \{s, s'\}$ then there is some $i \in [1, m - 1]$ such that $s = s_i$, $s' = s_{i+1}$, $\delta_\rho(s, \eta) = s'$ and $f(s) = f(s')$. Given these, we define a function g from N_R to the equivalence classes such that for all $n \in N_R$,

$$g(n) = [s] \quad \text{iff} \quad f(s) = n.$$

Recall that in Lemma 5.1, we have shown that given two PL expressions Q and Q' with their respective NFAs $M(Q)$ and $M(Q')$ then, $Q \subseteq Q'$ if and only if $S_Q \triangleleft S_{Q'}$. The symbols S_Q and $S_{Q'}$ are the start states of $M(Q)$ and $M(Q')$ respectively and \triangleleft is a simulation as defined in Definition 5.1. Furthermore, there is a function θ from N_Q to $N_{Q'}$ such that $\theta(S_Q) = S_{Q'}$, $\theta(F_Q) = F_{Q'}$, and for any state $s \in N_Q$, $s \triangleleft \theta(s)$. The symbols, N_Q and $N_{Q'}$, denote the sets of states in $M(Q)$ and $M(Q')$ respectively. Since $\rho \in P$, the language defined by ρ (which consists of a single string ρ) is contained in the language defined by P , i.e., $\rho \subseteq P$. Thus, there exists a function θ from N_ρ to N_P and a simulation relation \triangleleft such that $\theta(S_\rho) = S_P$, $\theta(F_\rho) = F_P$, and for any $s \in N_\rho$, $s \triangleleft \theta(s)$. It is easy to verify the following claim:

Claim: For all $s, s' \in [s]$, $\theta(s) = \theta(s')$.

Indeed, as observed earlier, if $s, s' \in [s]$, then there is some $i \in [1, m - 1]$ such that $s = s_i$, $s' = s_{i+1}$ and $\delta_\rho(s, \eta) = s'$. Since η does not appear in P , if $\theta(s) = n'$ and $\theta(s') = n''$, then there must be $\delta_P(n', _) = n''$ and $n' = n''$, by the definition of simulation relations. As a result, we can define $\theta([s])$ to be $\theta(s)$. Given these, to show $R \subseteq P$, it suffices to show that for any $n \in N_R$,

$$n \triangleleft \theta(g(n)).$$

For if it holds, then $S_R \triangleleft \theta(g(S_R)) = \theta(S_\rho) = S_P$. We next show that this holds. Assume, by contradiction, that there is $n \in N_R$ such that it is not the case that $n \triangleleft \theta(g(n))$. Let n be such a state with the largest index in the sequence of states in N_R starting from S_R .

Then by the definition of simulation relations given in Section 5.2.1, we must have one of the following cases.

(i) $n = F_R$ and either

1. $\theta(g(F_R)) \neq F_P$, or
2. $\theta(g(F_R)) = F_P$ but $\delta_R(F_R, -) = F_R$, $\delta_P(F_P, -) = \emptyset$.

The first case contradicts the assumption that $g(F_R) = [F_\rho]$ and $\theta([F_\rho]) = \theta(F_\rho) = F_P$. If it were the second case, then by $\delta_R(F_R, -) = F_R$, we would have $g(F_R) = \{F_\rho, s_{m-1}\}$ and $\delta_\rho(s_{m-1}, \eta) = F_\rho$. By the above claim, there must be $\theta(s_{m-1}) = \theta(F_\rho) = F_P$ and $\delta_P(F_P, -) = F_P$. Again this contradicts the assumption.

(ii) $n \neq F_R$ and either

1. $\delta_R(n, -) = n$ but $\delta_P(\theta(g(n)), -) \neq \theta(g(n))$, or
2. there is some label $l \in A$ such that $\delta_R(n, l) = n'$, but we have neither $\delta_P(\theta(g(n)), l) \neq \theta(g(n'))$ nor $\delta_P(\theta(g(n)), -) = \theta(g(n))$.

If it were the first case, then by the definition of the function g , we would have that $g(n) = \{s_i, s_{i+1}\}$ and $\delta_\rho(s_i, \eta) = s_{i+1}$. Thus by the above claim, there must be $\theta(s_i) = \theta(s_{i+1})$, $\delta_P(\theta(s_i), -) = \theta(s_i)$ and, in addition, $\theta(g(n)) = \theta(s_i)$. Hence $\delta_P(\theta(g(n)), -) = \theta(g(n))$, which contradicts the assumption. If it were the second case, then given $\delta_R(n, l) = n'$, we would have either $\delta_P(\theta(g(n)), l) = \theta(g(n'))$ or $\delta_P(\theta(g(n)), -) = \theta(g(n))$, by the definition of simulation relations and $g(n) \triangleleft \theta(g(n))$. Again this contradicts the assumption. Thus $n \triangleleft \theta(g(n))$ for all $n \in N_R$. \square

We are now in position to show that abstract trees have the following property:

Lemma 5.5 *Let $\Sigma \cup \{\varphi\}$ be a finite set of \mathcal{K} constraints. If there is a finite abstract tree T such that $T \models \Sigma$ and $T \models \neg\varphi$, then there is a finite XML tree G such that $G \models \Sigma$ and $G \models \neg\varphi$.*

Proof. Let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K} , and T be a finite abstract tree such that $T \models \Sigma$ and $T \not\models \varphi$. Let η be an element tag that does not occur in any key of $\Sigma \cup \{\varphi\}$, and G be the XML tree defined from T using η . We shall prove that $G \models \Sigma$ and $G \models \neg\varphi$. From Lemma 5.4, it follows immediately that for any path expression P in PL , $\llbracket P \rrbracket$ consists of the same nodes in T and G . For if $T \models P(r, a)$, where r is the root, then there is a path R in T such that $T \models R(r, a)$ and $R \subseteq P$. By Lemma 5.4, we have $G \models \rho(r, a)$, where ρ is the path w.r.t. R and $\rho \in P$. That is, a is in $\llbracket P \rrbracket$ in the tree G . Conversely, if a is in $\llbracket P \rrbracket$ in the tree G , then there is a path $\rho \in P$ such that $G \models \rho(r, a)$. Again by Lemma 5.4, $T \models R(r, a)$ and $R \subseteq P$, where R is the path expression w.r.t. ρ . Thus, a is in $\llbracket P \rrbracket$ in the abstract tree T .

We are now ready to show that $G \models \Sigma$ and $G \models \neg\varphi$. Suppose, by contradiction, that there exists a key $\phi = (Q, (Q', \{P_1, \dots, P_k\}))$ in Σ such that $G \models \neg\phi$. Then there exist a node $n \in \llbracket Q \rrbracket$, two distinct nodes $n_1, n_2 \in n\llbracket Q' \rrbracket$ and, in addition, for all $i \in [1, k]$, there exist nodes $x_i \in n_1\llbracket P_i \rrbracket$, $y_i \in n_2\llbracket P_i \rrbracket$ such that $x_i =_v y_i$. But by Lemma 5.4, we would have $T \models P_i(n_1, x_i) \wedge P_i(n_2, y_i)$ for all $i \in [1, k]$. Therefore, $T \models \phi$, which contradicts our assumption. We next show $G \models \neg\varphi$. Let $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. Since $T \models \neg\varphi$, there must exist a node $n \in \llbracket Q \rrbracket$, two distinct nodes $n_1, n_2 \in n\llbracket Q' \rrbracket$, and for all $i \in [1, k]$, there exist nodes x_i, y_i such that $x_i =_v y_i$ and, in addition, there exists a path R_i in T such that $T \models R_i(n_1, x_i) \wedge R_i(n_2, y_i)$, where $R_i \subseteq P_i$. Thus, by Lemma 5.4, there is path $\rho_i \in P_i$ such that $x_i \in n_1\llbracket \rho_i \rrbracket$, $y_i \in n_2\llbracket \rho_i \rrbracket$. Hence $G \models \neg\varphi$. \square

This property of abstract trees is now used to show that \mathcal{I} is a finite axiomatization for \mathcal{K} constraint implication.

Lemma 5.6 *The set \mathcal{I} is sound and complete for finite implication of \mathcal{K} constraints. That is, for any finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, $\Sigma \models \varphi$ if and only if $\Sigma \vdash_{\mathcal{I}} \varphi$.*

Proof. To simplify the discussion, we assume that all keys are in key normal form and all path expressions are in normal form. In general, given constraints ϕ and ϕ' in \mathcal{K} , where ϕ' is the key normal form of ϕ , ϕ and ϕ' are equivalent. That is, for any XML tree T , $T \models \phi$ iff $T \models \phi'$. Thus, the assumption does not lose generality.

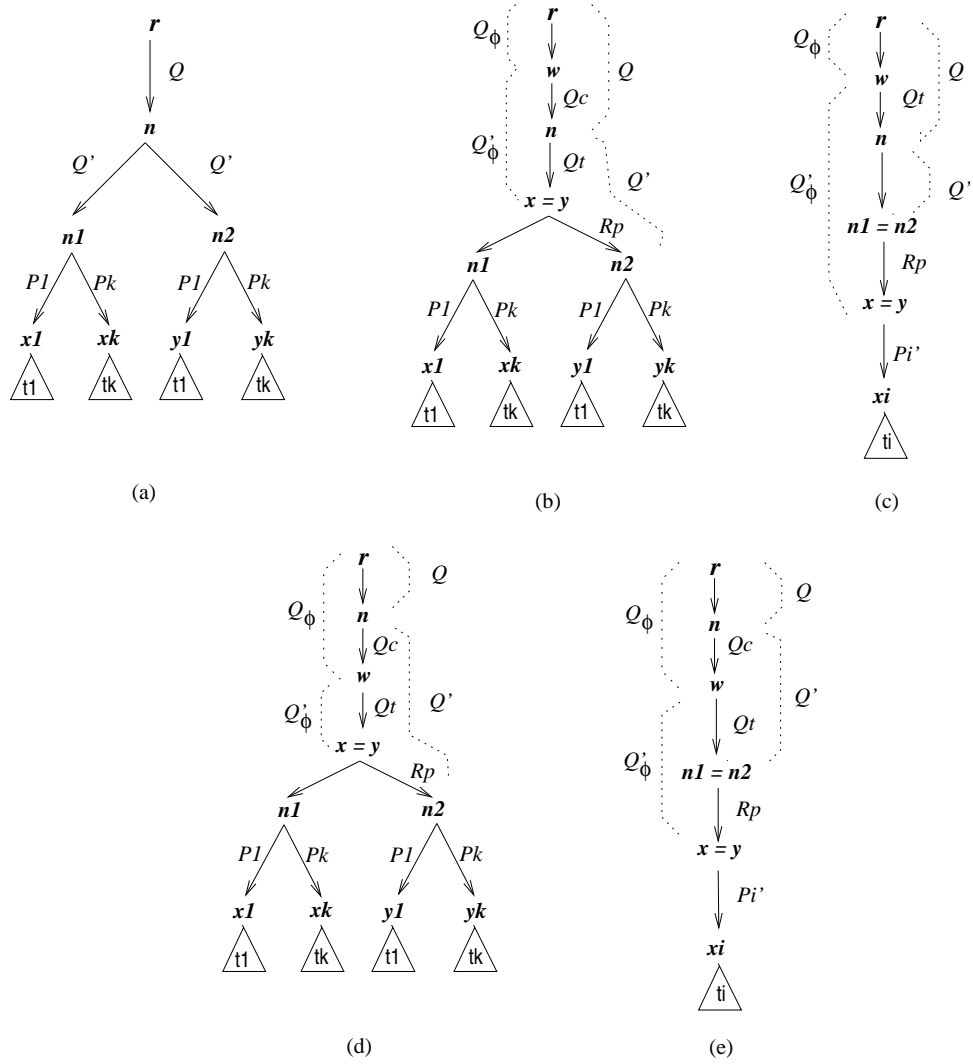


Figure 5.4: Abstract trees constructed in the proof of Lemma 5.6

Soundness of \mathcal{I} can be verified by induction on the lengths of \mathcal{I} -proofs. For the proof of completeness, let $\Sigma \cup \{\varphi\}$ be a finite set of keys in \mathcal{K} , where $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$. We show that if $\Sigma \not\vdash_{\mathcal{I}} \varphi$ then $\Sigma \not\models \varphi$. More specifically, assume that $\Sigma \not\vdash_{\mathcal{I}} \varphi$. Then we show that there exists an XML tree G such that $G \models \Sigma$ but $G \not\models \varphi$.

The construction of G involves the following steps: First, we define a finite abstract tree T such that $T \not\models \varphi$. Then, T is modified in a way that the resulting tree T_f satisfies Σ . That is, for each key $\phi \in \Sigma$, we check whether T satisfies ϕ . If not, certain nodes in T are merged such that the modified tree satisfies ϕ . At the end of the merging process,

$T_f \models \Sigma$ and either: (1) $T_f \not\models \varphi$, and by Lemma 5.5, we can construct an XML tree G from T_f that satisfies Σ but not φ ; or (2) $T_f \models \varphi$. In this case, we show that our assumption that $\Sigma \not\vdash_{\mathcal{I}} \varphi$ does not hold. That is, we show that each step of the merging operations corresponds to applications of certain rules in \mathcal{I} . Therefore, if $T_f \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts the assumption.

Before describing the construction of the abstract tree, we first modify the key φ if it is of the form $(Q, (Q'/@a, \{\}))$ to be $(Q, (Q', \{\}))$, since if we prove the latter, we have $\Sigma \vdash_{\mathcal{I}} \varphi$ by the *attribute* rule in \mathcal{I} . We also assume that $Q' \neq \epsilon$, since otherwise we have $\Sigma \vdash_{\mathcal{I}} \varphi$ by the *epsilon* rule.

The construction of a finite abstract T that does not satisfy φ is as follows. The abstract tree T consists of a single path Q from the root leading to a node n , which has two distinct subtrees T_1 and T_2 . Each subtree has a Q' path. These Q' paths lead to nodes n_1 and n_2 from n in T_1 and T_2 , respectively. From each of n_1 and n_2 there are paths P_1, \dots, P_k , as depicted in Figure 5.4 (a). For each $i \in [1, k]$, let x_i be the (single) node at the end of the P_i path in T_1 , and y_i be the (single) node at the end of the P_i path in T_2 .

Assume that for each $i \in [1, k]$, $x_i =_v y_i$, but for any other pair x, y in T , $x \neq_v y$. This can be achieved as follows: for each element in T we add a new text subelement. For any x, y in T , if they are x_i, y_i then we let them have the same value when they are A or S nodes, and let their text subelements have the same value when they are E nodes (in this case the text subelements are their only subelements). If they are not x_i, y_i then we let them have different values if they are A or S nodes, and let their text subelements have different values if they are E nodes. The only exception is when there is $i \in [1, k]$ such that $P_i = \epsilon$. In this case, we have to assure $n_1 =_v n_2$. That is, for all $j \in [1, k]$ and for any P'_j such that $P_j = P'_j/P''_j$ for some $P''_j \in PL$, we let $x'_j =_v y'_j$, where x'_j, y'_j are the nodes in $n_1[[P'_j]]$ and $n_2[[P'_j]]$, respectively. For any other pair x, y in T , we let $x \neq_v y$ as before. It is easy to see that $T \models \neg\varphi$.

We next modify T such that $T \models \Sigma$. Using the following algorithm and starting with T constructed above, we examine each ϕ in Σ . If the abstract tree does not satisfy ϕ , then we merge certain nodes in the tree such that the modified tree satisfies ϕ . Assume that for

each ϕ in Σ , $\phi = (Q_\phi, (Q'_\phi, \{P'_1, \dots, P'_m\}))$.

repeat until no further change in T

if there exist key $\phi \in \Sigma$ and nodes x, x'_1, \dots, x'_m in T_1 , y, y'_1, \dots, y'_m in T_2 ,
and node w in T such that

$$T \models Q_\phi(r, w) \wedge Q'_\phi(w, x) \wedge Q'_\phi(w, y) \wedge P'_1(x, x'_1) \wedge \dots \wedge P'_m(x, x'_m) \wedge \\ P'_1(y, y'_1) \wedge \dots \wedge P'_m(y, y'_m) \wedge x'_1 =_v y'_1 \wedge \dots \wedge x'_m =_v y'_m \wedge x \neq y$$

then merge x, y and their ancestors in T as follows:

Case 1: if x, y are on Q' paths from n to n_1, n_2
respectively, and they are not n_1, n_2

then merge nodes as shown in Figure 5.4 (b) and (d)

Case 2: if x, y are on some P_i in T_1, T_2 , respectively, or if they are n_1, n_2

then (i) merge nodes as shown in Figure 5.4 (c) and (e)

(ii) terminate the algorithm

By the construction of T , $x'_i =_v y'_i$ iff they are corresponding nodes in T_1 and T_2 , respectively. Moreover, the node w can only be either on path Q or on path Q' . In Case 1, the subtree under x and the subtree under y will both be under the same node $x = y$, as shown in Figure 5.4 (b) and (d). In Case 2, under the node n_1 (which is merged with n_2) only a single copy of the P_i path is retained and we discard the rest of the key paths in $\{P_1, \dots, P_k\}$. If x and y are n_1 and n_2 , respectively, a single copy of each of the P_i paths are retained under node n_1 .

The algorithm terminates since T is finite and thus merging can be performed only finitely many times. Let T_f denote the tree obtained upon the termination of the algorithm. Note that $T_f \not\models \varphi$ iff $n_1 \neq n_2$. If this is the case, by Lemma 5.5, there is an XML tree G such that $G \models \Sigma$ and $G \not\models \varphi$, which completes the proof. On the other hand, if $T_f \models \varphi$, that is, the algorithm terminates in Case 2, we have to show that $\Sigma \vdash_{\mathcal{I}} \varphi$, which leads to a contradiction.

Let us also use T to denote the tree obtained after executing z merging operations. We show by induction on z that each step of merging corresponds to applications of certain

rules of \mathcal{I} , and thus if $T \models \varphi$ then $\Sigma \vdash_{\mathcal{I}} \varphi$. For the base case, $z = 0$, the statement holds since the initial tree does not satisfy φ . Assume the statement for z . We will show that it also holds for $z + 1$.

First, consider the merging in Case 1 as shown in Figure 5.4 (b) and (d). This step generates \mathcal{I} -proofs for keys that will be used in establishing $\Sigma \vdash_{\mathcal{I}} \varphi$ if $T_f \models \varphi$. By the definition of abstract trees, Case 1 can only happen if there is a PL expression R_p such that $Q/Q' \subseteq Q_\phi/Q'_\phi/R_p$ and in addition, for all $j \in [1, m]$, there is $s \in [1, k]$ such that either (i) $R_p/P_s \subseteq P'_j$ or (ii) there is a PL expression R_j such that $R_p/P_s \subseteq P'_j/R_j$. If it is (ii) then there must exist some $l \in [1, k]$ such that $P_l = \epsilon$ in φ , by the definition of T . We consider the following cases.

(a) If the node w is on the path Q , i.e., it is above n in T , then there must be PL expression Q_t such that $Q' = Q_t/R_p$, and $x, y \in n[[Q_t]]$ as illustrated in Figure 5.4 (b). Moreover, from ϕ the following can be proved:

$$(Q, (Q_t, \{R_p/P_1, \dots, R_p/P_k\}))$$

by using *target-to-context*, three *containment* rules (i.e., *context-path-containment*, *target-path-containment* and *key-path-containment*) and *superkey*. If it is (ii) then *prefix-epsilon* is also needed.

(b) If the node w is on the path Q' , i.e., it is below n but above n_1, n_2 in T , then there must be PL expressions Q_c, Q_t such that $Q/Q_c \subseteq Q_\phi$, $Q' = Q_c/Q_t/R_p$, $w \in n[[Q_c]]$ and $x, y \in n[[Q_c/Q_t]]$ as illustrated in Figure 5.4 (d). This can only happen when some descendants x', y' of n on path Q' above x, y were merged in a previous step by the algorithm. More precisely, there are PL expressions Q_{t1}, Q_{t2} such that $Q_t = Q_{t1}/Q_{t2}$, $x', y' \in n[[Q_c/Q_{t1}]]$ and x', y' were merged in Case 1 of the algorithm. Thus by the induction hypothesis, we have that the following is provable from Σ by using \mathcal{I} :

$$(Q, (Q_c/Q_{t1}, \{Q_{t2}/R_p/P_1, \dots, Q_{t2}/R_p/P_k\})).$$

From ϕ the following can be proved

$$(Q/Q_c, (Q_{t1}/Q_{t2}, \{R_p/P_1, \dots, R_p/P_k\}))$$

by using the three *containment* rules and *superkey*. If it is (ii) then *prefix-epsilon* is also needed. Thus by *target-to-context* and *interaction* we have

$$(Q, (Q_c/Q_{t1}/Q_{t2}, \{R_p/P_1, \dots, R_p/P_k\})).$$

That is, $(Q, (Q_c/Q_t, \{R_p/P_1, \dots, R_p/P_k\}))$.

(2) Next, we consider the merging in Case 2 as shown in Figure 5.4 (c) and (e). If it is the case then we show $\Sigma \vdash_{\mathcal{I}} \varphi$. By the definition of abstract trees, Case 2 can only happen if there is a *PL* expression R_p such that $Q/Q'/R_p \subseteq Q_\phi/Q'_\phi$ and in addition, for all $j \in [1, m]$, there is $s \in [1, k]$ such that either (i) $P_s \subseteq R_p/P'_j$ or (ii) there is a *PL* expression R_j such that $P_s \subseteq R_p/P'_j/R_j$. If it is (ii) then there must exist some $l \in [1, k]$ such that $P_l = \epsilon$ in φ , by the definition of T . We consider the following cases.

(a) If the node w is on the path Q , i.e., it is above n in T , then there must be *PL* expression Q_t such that $Q_t/Q'/R_p \subseteq Q'_\phi$, $x \in n_1[[R_p]]$ and $y \in n_2[[R_p]]$ as illustrated in Figure 5.4 (c). If $R_p = \epsilon$ then φ can be proved from ϕ by using *target-to-context*, the three *containment* rules and *superkey*. Note that if it is (ii) then *prefix-epsilon* is also needed. If $R_p \neq \epsilon$ then by the construction of T , we must have $m = 1$. Thus, we can also prove φ from ϕ by using *subnodes*, *target-to-context*, the three *containment* rules and *superkey*. Thus, we have $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption.

(b) If the node w is on the path Q' , i.e., it is below n but above n_1, n_2 in T , then there must be *PL* expressions Q_c, Q_t such that $Q/Q_c \subseteq Q_\phi$, $Q' = Q_c/Q_t$, $w \in n[[Q_c]]$, $x \in n_1[[R_p]]$ and $y \in n_2[[R_p]]$ as illustrated in Figure 5.4 (e). This can only happen when some descendants x', y' of n on path Q' above n_1, n_2 were merged in a previous step by the algorithm. More precisely, there are *PL* expressions Q_{t1}, Q_{t2} such that $Q_t = Q_{t1}/Q_{t2}$, $x', y' \in n[[Q_c/Q_{t1}]]$ and x', y' were merged in Case 1 of the algorithm. Thus, by the induction hypothesis, we have that the following is provable from Σ by using \mathcal{I} :

$$(Q, (Q_c/Q_{t1}, \{Q_{t2}/P_1, \dots, Q_{t2}/P_k\})).$$

If $R_p = \epsilon$ then from ϕ the following can be proved

$$(Q/Q_c, (Q_{t1}/Q_{t2}, \{P_1, \dots, P_k\}))$$

by using the three *containment* rules and *superkey*. Observe that if it is (ii) then *prefix-epsilon* is also needed. If $R_p \neq \epsilon$ then by the construction of T , we must have $m = 1$. Thus, we can also prove it from ϕ by using *subnodes*, *target-to-context*, the three *containment* rules and *superkey*. Thus by *interaction* and *target-to-context* we have

$$(Q, (Q_c/Q_{t1}/Q_{t2}, \{P_1, \dots, P_k\})).$$

That is, $(Q, (Q', \{P_1, \dots, P_k\})) = \varphi$. Thus again we have $\Sigma \vdash_{\mathcal{I}} \varphi$, which contradicts our assumption.

This shows that \mathcal{I} is complete for \mathcal{K} constraint implication and thus completes the proof of Lemma 5.6. \square

Finally, we show that \mathcal{K} constraint implication is decidable in polynomial time.

Lemma 5.7 *There is an algorithm that, given any finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, determines whether $\Sigma \models \varphi$ in time $O(n^7)$, where n is the size of keys.*

Proof. A function for determining finite implication of \mathcal{K} constraints is given in Algorithm 5.5.

The correctness of the algorithm follows from Lemma 5.6 and its proof. Similar to the algorithm for implication of absolute keys (Figure 5.3), it applies \mathcal{I} rules to derive φ if $\Sigma \models \varphi$. Observe that Steps 1, 2, and 3 are identical in both algorithms, and Step 5(a), when ignoring the context path, proves φ from ϕ by applying exactly the same inference rules as the algorithm in Figure 5.3. In fact, if we replace paths Q , Q_ϕ , and Q_t by ϵ in Step 5(a), it is identical to the algorithm in Figure 5.3.

The presence of a context path adds complexity to the algorithm for two reasons. First, it can be the case that either the context path of a key in Σ is contained in a prefix of Q , considered in Steps 5(a) and (c); or Q is contained in a prefix of the context path of a key in Σ , considered in Steps 5(b) and (d). Second, the application of the *interaction* rule depends on the existence of two distinct keys in Σ . As a consequence, we need to keep track of intermediate keys in the \mathcal{I} -proof. In the algorithm, these keys are produced by Steps 5(c) and (d), and they are kept in the set variable X .

Algorithm implication

Input: a finite set $\Sigma \cup \{\varphi\}$ of \mathcal{K} constraints, where $\varphi = (Q, (Q', \{P_1, \dots, P_k\}))$

Output: true iff $\Sigma \models \varphi$

1. if $\varphi = (Q, (Q'/@l, \{\}))$ then $\varphi := (Q, (Q', \{\}))$;
2. if $Q' = \epsilon$ or $Q' = @l$ then output true and terminate
3. for each $(Q_i, (Q'_i, S_i)) \in \Sigma \cup \{\varphi\}$ do
 repeat until no further change
 if $S_i = S \cup \{P', P''\}$ such that $P' \subseteq P''$ then $S_i := S_i \setminus \{P''\}$
4. $X := \emptyset$;
5. repeat until no keys in Σ can be applied in cases (a)-(d).
 for each $\phi = (Q_\phi, (Q'_\phi, \{P'_1, \dots, P'_m\})) \in \Sigma$ do
 // See Figure 5.4(c) for an illustration of this case.
 (a) if there is Q_t, R_p in PL such that $Q \subseteq Q_\phi/Q_t$, $Q_t/Q'/R_p \subseteq Q'_\phi$, $R_p = \epsilon$ if $m > 1$
 and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 (i) $P_s \subseteq R_p/P'_j$ or
 (ii) $P_s = R'_s/R''_s$, $R'_s \subseteq R_p/P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then output true and terminate
 // See Figure 5.4(e) for an illustration of this case.
 (b) if there are Q_c, Q_t, R_p in PL such that
 $Q/Q_c \subseteq Q_\phi$, $Q'/R_p \subseteq Q_c/Q'_\phi$, $Q' = Q_c/Q_t$, $R_p = \epsilon$ if $m > 1$, and
 for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 (i) $P_s \subseteq R_p/P'_j$ or
 (ii) $P_s = R'_s/R''_s$, $R'_s \subseteq R_p/P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$;
 and moreover, there is $(Q, (Q_c, \{Q_t/P_1, \dots, Q_t/P_k\}))$ in X
 then output true and terminate
 // See Figure 5.4(b) for an illustration of this case.
 (c) if there are Q_c, Q_t, R_p in PL such that $Q \subseteq Q_\phi/Q_c$, $Q_c/Q' \subseteq Q'_\phi/R_p$, $Q' = Q_t/R_p$
 and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 (i) $R_p/P_s \subseteq P'_j$ or
 (ii) $P_s = R'_s/R''_s$, $R_p/R'_s \subseteq P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$
 then
 (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1, \{Q_2/R_p/P_1, \dots, Q_2/R_p/P_k\}))\}$
 where $Q_t = Q_1/Q_2$ for some $Q_1, Q_2 \in PL$;
 (2) if $m > 1$ then $X := X \cup \{(Q, (Q_t, \{R_p/P_1, \dots, R_p/P_k\}))\}$;
 (3) $\Sigma := \Sigma \setminus \{\phi\}$;
 // See Figure 5.4(d) for an illustration of this case.
 (d) if there are Q_c, Q_t, R_p in PL such that $Q/Q_c \subseteq Q_\phi$, $Q' \subseteq Q_c/Q'_\phi/R_p$,
 $Q' = Q_c/Q_t/R_p$ and for all $j \in [1, m]$ there is $s \in [1, k]$ such that either
 (i) $R_p/P_s \subseteq P'_j$ or
 (ii) $P_s = R'_s/R''_s$, $R_p/R'_s \subseteq P'_j$, and there exists $l \in [1, k]$ such that $P_l = \epsilon$;
 and moreover, there is $(Q, (Q_c, \{Q_t/R_p/P_1, \dots, Q_t/R_p/P_k\}))$ in X
 then
 (1) if $m = 1$ then $X := X \cup \{(Q, (Q_1, \{Q_2/R_p/P_1, \dots, Q_2/R_p/P_k\}))\}$
 where $Q_c/Q_t = Q_1/Q_2$ for some $Q_1, Q_2 \in PL$;
 (2) if $m > 1$ then $X := X \cup \{(Q, (Q_c/Q_t, \{R_p/P_1, \dots, R_p/P_k\}))\}$;
 (3) $\Sigma := \Sigma \setminus \{\phi\}$;
6. output false

Figure 5.5: Finite implication of \mathcal{K} constraints

Next, observe that each conditional statement in step 5 corresponds to applications of certain rules in \mathcal{I} . More specifically:

- Steps 5(a) and (c) use the three *containment* rules (i.e., *context-path-containment*, *target-path-containment* and *key-path-containment*), *target-to-context*, *superkey*, and *subnodes*. If it is (ii) then *prefix-epsilon* is also used.
- Steps 5(b) and (d) apply the three *containment* rules, *superkey*, *subnodes*, and *interaction*, which need intermediate results of the \mathcal{I} -proof stored in X . If it is (ii) then *prefix-epsilon* is also used.

For the interested reader, Step 5(a) corresponds to Figure 5.4(c). Since nodes n_1 and n_2 are merged when considering a key $\phi \in \Sigma$, we can prove φ . Similarly, Step 5(b) corresponds to Figure 5.4(e). The difference between Steps 5(a) and (b) is whether or not the context path of the key contains a prefix of Q . Steps 5(c) and (d) correspond to Figure 5.4(b) and (d) respectively. Here these keys do not prove φ directly, but they generate intermediate results, which are saved in X . Again the difference is whether the context path of the key contains a prefix of Q .

We next show that this algorithm runs in polynomial time. To see this, observe that Steps 1 and 2 take constant time and Step 3 takes at most $O((|\Sigma| + |\varphi|)^3)$ time. For Step 5, the worst scenario can happen as follows: for each key in Σ , the conditions of (a) - (d) are tested and only the last key in Σ is removed after testing all keys in Σ . Hence, the second time the for loop is performed, one less key is tested. Therefore if there are s keys in Σ , a total of $O(s^2)$ keys will be tested. We next examine the complexity of each condition of Steps (a) - (d). For Step (a), we need to partition Q to find Q_t . Also, for each such Q_t , we need to partition Q'_ϕ to find R_p . Since containment of path expressions is tested in quadratic time, the first two inclusion tests cost at most $|Q| * (|\varphi| * (|\phi| + |\varphi|) + |Q'_\phi| * ((|\varphi| + |\phi|) * |\phi|))$, which is $O(n^4)$ in total, where n is the size of keys. Then for each key path P'_j in ϕ , we check if there is a key path P_s in φ and partition P_s to get R'_s such that case (i) or (ii) is satisfied. This costs $|P_s| * (|R_p| + |P'_j|) + |P_s| * |P_s| * (|R_p| + |P'_j|)$. Since there are m key paths in ϕ , for all k key paths in φ these tests cost $(|P_1| + \dots + |P_k|) * (m * |R_p| + |P'_1| +$

$\dots + |P'_m|) + (|P_1| * |P_1| + \dots + |P_k| * |P_k|)(m * |R_p| + |P'_1| + \dots + |P'_m|)$. Note that $R_p = \epsilon$ when $m > 1$, and there are $|Q| * |Q'_\phi|$ possible expressions for Q_t , and R_p . Therefore, the cost of step (a) is at most $|Q| * |Q'_\phi| * (|\varphi| * (|\phi| + |\phi|) + |\varphi|^2 * (|\phi| + |\phi|))$, which is $O(n^5)$. It is easy to see that the Steps (b), (c), and (d) involve at most the same cost. Since these tests are performed $O(s^2)$ times, the overall cost of the algorithm is $O(n^7)$, and therefore we have a polynomial algorithm. It is possible that this algorithm can be improved further to achieve a lower complexity but this is beyond the scope of this work. \square

5.2.4 Axiomatization for Strong Key Implication

In this Section we will discuss the problem of finite implication for \mathcal{K}_{att} . Recall that the differences of keys of \mathcal{K}_{att} from those of \mathcal{K} are twofold: key paths are restricted to be simple attributes, and they are required to exist. In Table 5.4 we give a set of inference rules for \mathcal{K}_{att} , denoted as \mathcal{I}_{att} . Rules context-containment, target-containment, target-to-context, and epsilon are identical to the ones shown in Table 5.3, while the *uniqueness* rule has no counterpart there. The superkey, context-to-target, and attribute rules are restrictions of ones shown in Table 5.3. We briefly illustrate these rules below.

- *superkey*: if a set of attributes S is a key, then more attributes can be added to S , as long as they are required to exist.
- *context-to-target*: for any node n in $\llbracket Q \rrbracket$, there is at most one node n' in $n\llbracket Q_1 \rrbracket$; thus $n\llbracket Q_1/Q_2 \rrbracket = n'\llbracket Q_2 \rrbracket$. As a result, if S can uniquely identify a node in $n'\llbracket Q_2 \rrbracket$, it can also uniquely identify a node in $n\llbracket Q_1/Q_2 \rrbracket$. Observe that this rule restricts the interaction rule of \mathcal{I} for the case when key paths are simple attributes. That is, in \mathcal{I} , if $(Q, (Q_1, \{\}))$ then by the superkey rule we can obtain $(Q, (Q_1, \{Q_2/S\}))$. Given the key $(Q/Q_1, (Q_2, S))$, by interaction we obtain $(Q, (Q_1/Q_2, S))$.
- *uniqueness*: if for every node n in $\llbracket Q \rrbracket$ there exists at most a single node in $n\llbracket Q_1/@l \rrbracket$, and every node n' in $n\llbracket Q_1 \rrbracket$ is required to have attribute $@l$ (since $@l \in S$) then it must be the case that there exists at most one node in $n\llbracket Q_1 \rrbracket$. This is the only rule that is a direct consequence of key paths being required to exist under certain nodes.

$\frac{(Q, (Q', S)), (Q/Q', (\epsilon, S')), @l \in S'}{(Q, (Q', S \cup \{@l\}))}$	(superkey)
$\frac{(Q, (Q', S)), Q_1 \subseteq Q}{(Q_1, (Q', S))}$	(context-containment)
$\frac{(Q, (Q', S)), Q_2 \subseteq Q'}{(Q, (Q_2, S))}$	(target-containment)
$\frac{(Q, (Q_1/Q_2, S))}{(Q/Q_1, (Q_2, S))}$	(target-to-context)
$\frac{(Q, (Q_1, \{\}), (Q/Q_1, (Q_2, S)))}{(Q, (Q_1/Q_2, S))}$	(context-to-target)
$\frac{(Q, (Q_1/@l, \{\})), (Q, (Q_1, S)), @l \in S}{(Q, (Q_1, \{\}))}$	(uniqueness)
$\frac{}{(Q, (@l, \{\}))}$	(attribute)
$\frac{}{(Q, (\epsilon, \{\}))}$	(epsilon)

Table 5.4: \mathcal{I}_{att} : Rules for strong key implication

- *attribute*: any path expression Q has at most one attribute labeled $@l$. Observe that this rule is simpler, but equivalent to the one shown for keys in \mathcal{K} .

Based on the inference rules in \mathcal{I}_{att} , we develop Algorithm `implication` shown in Figure 5.6 for checking implication of keys in \mathcal{K}_{att} . The algorithm is based on the following observation: given a key $\phi = (Q, (Q', S))$, it is *not* the case that new keys can be derived from ϕ by pushing down part of the context path Q to its target, but it is always possible to pull up part of the target path Q' to its context. Intuitively, we can always derive keys that are local to subtrees given that they hold on larger trees, but not the other way around. Therefore, “local” keys are easier to check. Based on this observation, the algorithm for checking whether a key $\varphi = (Q, (Q', S))$ is provable from Σ can be divided in two parts. First, φ is modified to be “as local as possible”, by pulling up the target path Q' to its context, such that the resulting key φ' satisfies the following condition: if $\Sigma \models \varphi'$ then $\Sigma \models \varphi$. It is based on a reverse use of the *context-to-target* rule. That is, given that

$Q' = Q_1/Q_2$, and $\Sigma \models (Q, (Q_1, \{\}))$, if $\Sigma \models \varphi'$, where $\varphi' = (Q/Q_1, (Q_2, S))$, then $\Sigma \models \varphi$ by the *context-to-target* rule. Observe that the modification on φ can only be executed if $\Sigma \models (Q, (Q_1, \{\}))$. Therefore, in the algorithm, before modifying the key to be checked (Lines 4 to 6), Σ is extended by the *uniqueness* rule (Lines 1 to 3) to contain all keys of the form $(Q, (Q', \{\}))$ derivable from Σ .

The second part of the algorithm checks whether φ' is provable from Σ . Since the *context-to-target* rule has already been applied and all keys that can be derived by the *uniqueness* rule have been inserted in Σ , $\Sigma \models \varphi'$, where $\varphi' = (Q_1, (Q'_1, S))$, if and only if it is one of the following cases:

- (1) the target path Q'_1 is a single attribute label (Line 7) and $\Sigma \models \varphi'$ by the *attribute* rule;
- (2) the target path $Q'_1 = \epsilon$ and all attributes in S are required to exist (Line 8); in this case, $\Sigma \models \varphi'$ by the *epsilon* rule. Observe that Function `exist` is invoked for checking the existence of attributes in S ;
- (3) there exists a key $\phi = (Q_2, (Q'_2, S_2))$ in Σ such that
 - (a) $S_2 \subseteq S_1$, and attributes in $(S_1 \setminus S_2)$ are required to exist (Line 10), and moreover,
 - (b) context path Q_1 of φ' can be partitioned into P_1/P'_1 such that $P_1 \subseteq Q_2$ and $P'_1/Q'_1 \subseteq Q'_2$ (Line 11).

Condition (a) checks whether φ' can be derived from ϕ by the *superkey* rule and condition (b) checks for the applicability of the *target-to-context*, *context-containment*, and *target-containment* rules.

Therefore, the algorithm verifies if each of the rules in \mathcal{I}_{abs} can be applied in proving that $\Sigma \models \varphi$. We now state our main result for keys in \mathcal{K}_{att} .

Lemma 5.8 *For determining finite implication of keys of \mathcal{K}_{att} ,*

- *the set \mathcal{I}_{att} is sound and complete; and*
- *there is an $O(n^4)$ time algorithm, where n is the length of constraints involved.*

Proof. We omit the proof of soundness and completeness of \mathcal{I}_{att} because most of the proof can be verified along the same lines as the proof of Lemma 5.6 discussed in Section 5.2.3.

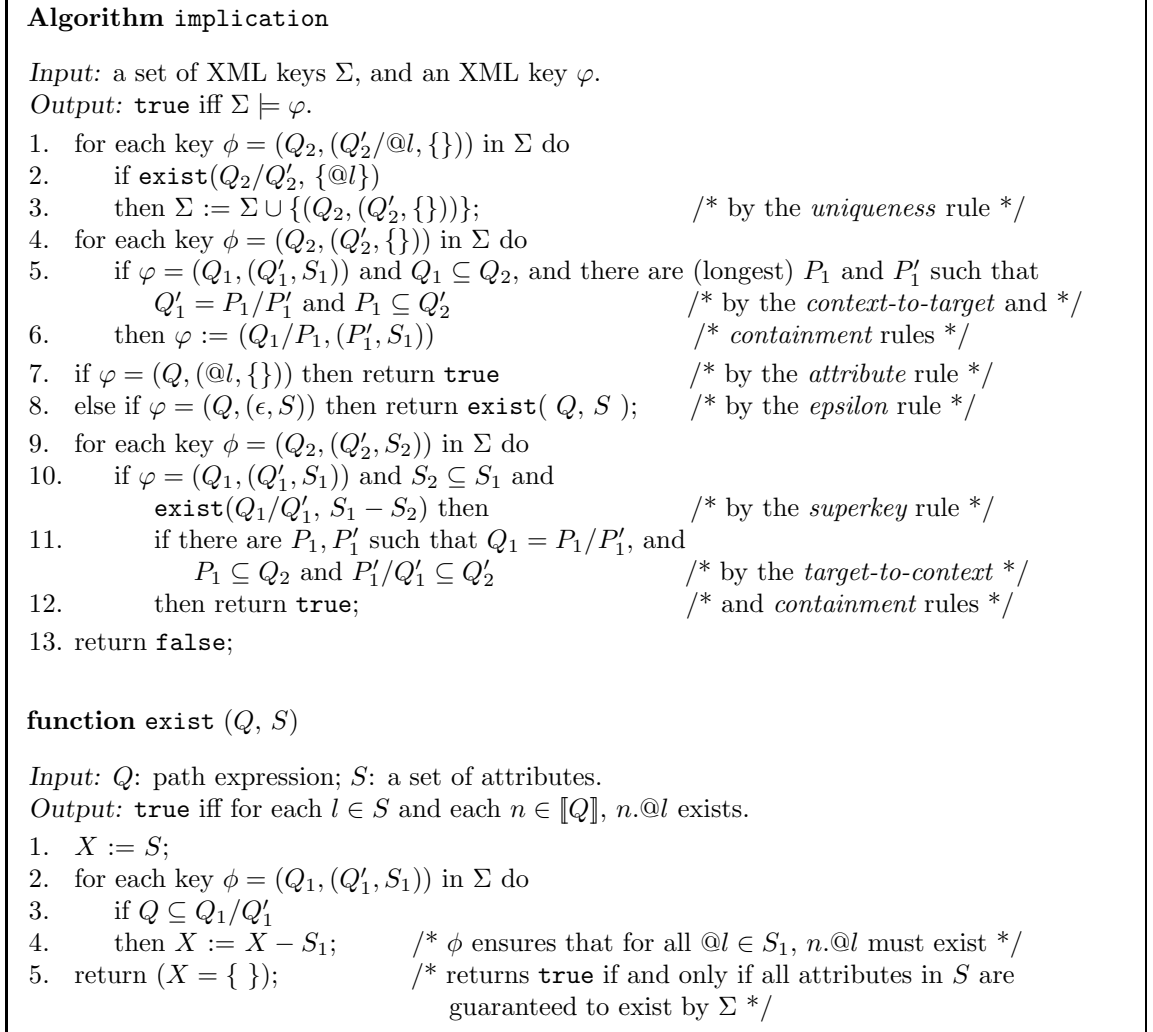


Figure 5.6: Finite implication of \mathcal{K}_{att} constraints

A formal proof of correctness of the algorithm in Figure 5.6 is long yet simple; it follows from the axioms for constraint implication and thus omitted. We will now show that the algorithm takes $O(|\Sigma|^2 |\varphi|^2)$ time, where $|\Sigma|$ and $|\varphi|$ are the sizes of Σ and φ , respectively. Indeed, in Section 5.2.1 we have presented an algorithm for checking if a path expression P_1 is contained in P_2 which is quadratic, $O(|P_1||P_2|)$. Thus Function **exist** can be done in $O(|\Sigma||Q|)$ time. Given the cost of Function **exist**, it is easy to see that Lines 1 to 3 of algorithm **propagation** take at most $O(|\Sigma|^2)$ time. Line 5 takes at most $O(|Q'_1| (|Q_1||Q_2| + |P_1||Q'_2|))$ since there exists $|Q'_1|$ ways of partitioning Q'_1 in P_1 and P'_1 . Therefore, for all keys in Σ , Lines 4 to 6 take at most $O(|\varphi| (|\varphi||\Sigma| + |\varphi||\Sigma|))$ time, that is $O(|\varphi|^2|\Sigma|)$.

For Line 10, recall that the key paths in S_2 and S_1 are simple attributes, and therefore the containment of S_2 in S_1 can be checked in $O(|S_2||S_1|)$ time. Function `exist` is in $O(|\Sigma|(|Q_1| + |Q'_1|))$ time. Thus, Lines 10 to 11 take at most $O(|S_2||S_1| + |\Sigma|(|Q_1| + |Q'_1|) + |Q_1| (|P_1||Q_2| + (|P'_1| + |Q'_1|)|Q'_2|))$. Hence for all keys in Σ , Lines 9 to 12 cost at most $O((|\Sigma||\varphi|) + |\Sigma||\varphi| + |\varphi| (|\varphi||\Sigma| + |\varphi||\Sigma|))$ time, which again is $O(|\Sigma| |\varphi|^2)$. Therefore, the whole algorithm takes $O(|\Sigma|^2 + |\Sigma| |\varphi|^2)$ time, or simply $O(|\Sigma|^2 |\varphi|^2)$. \square

5.3 Discussion

We have investigated two key constraint languages for XML, and studied the associated (finite) satisfiability and (finite) implication problems in the absence of DTDs. These keys are capable of expressing many important properties of XML data. Moreover, in contrast to other proposals, keys defined in these languages can be reasoned about efficiently. More specifically, keys expressed in these languages are always finitely satisfiable, and their (finite) implication is finitely axiomatizable and decidable in PTIME in the size of keys. We believe that these key constraints are simple yet expressive enough to be adopted by XML designers and maintained by systems for XML applications.

One might be interested in using different path languages to express keys. The containment and equivalence problems for the full regular language are PSPACE-complete [GJ79], and they are not finitely axiomatizable. Another alternative is to adopt the language of [MS99], which simply adds a single wildcard to PL . Despite the seemingly trivial addition, containment of expressions in their language is only known to be in PTIME. It would be interesting to develop an algorithm for determining containment of expressions in this language with a complexity comparable to the related result established in this work. For XPath [CD99] expressions, it has been shown [MS04, NS03] that it is rather expensive to determine containment of XPath expressions.

Along the same lines as our XML key language, a language of foreign keys needs to be developed for XML. As shown by [FS03, FL02, AFL02a], the implication and finite implication problems for a class of keys and foreign keys defined in terms of XML attributes

are undecidable, in the presence or absence of DTDs. However, under certain practical restrictions, these problems are decidable in PTIME. Whether these decidability results still hold for more complex keys and foreign keys needs further investigation.

There are more general definitions of key constraints than the ones considered in this Chapter that deserve further investigation. Among them are keys that imply value-equality on nodes rather than node identity, as described in 4.2.3. This is useful in XML documents in which redundancy is tolerated. It is sometimes useful to put the same information in more than one place in an XML document in order to avoid having to do joins to recover this information. Definitions of functional dependencies for XML (XFDs)[AL04, VLL04, CDHZ03], described in Section 4.4, are also capable of expressing such redundancies. Like in relational databases, XFDs of [AL04] and [VLL04] form the basis to define a normal form for XML documents. In contrast, the purpose of defining XFDs in [CDHZ03] is to reduce the redundancy of data when using relational technology to store an XML document. A sound set of inference rules are given, as well as an implication algorithm based on these rules. But the rules are not proven to be complete. A different approach for validating a relational schema for storing XML data is to compute the set of constraints that are proven to be valid in the relational storage given that certain constraints are satisfied by the XML document. These relational constraints can then be used to check whether the XML data being stored is compatible with an existent schema or to design a new relational schema. This approach is explored in the next Chapter. That is, based on our ability to reason about keys efficiently, we will develop algorithms for computing a complete set of functional dependencies that are proven to be satisfied by the relational storage of XML data, given that the document satisfies a set of XML keys.

Chapter 6

Propagating XML Constraints to Relations

A common paradigm in many application areas in which XML is used as a data exchange format is for a data provider to export its data using XML; on the other end, the data consumer imports some or all of the XML data and stores it using database technology. Since the XML data being transmitted is often large in size and fairly regular in structure, the database technology used is frequently relational.

A recognized problem with XML is that it is only syntax and does not carry the semantics of the data. To address this problem, a number of constraint specifications have recently been proposed for XML which include a notion of keys, presented in Chapter 4. A natural question to ask, therefore, is how information about constraints can be used to determine when an existing consumer database design is incompatible with the data being imported, or to generate de-novo a good consumer database.

For example, the GUS database [DCB⁺01] at the Penn Center for Bioinformatics imports XML gene expression data (MAGE-ML) and maps portions of it into a pre-existing relational schema using a Perl object layer. Due to the complexity of the GUS schema, the MAGE-ML data, and the transformation, it is very difficult to know whether or not inconsistency (integrity constraint errors) will arise as the data is imported. We illustrate

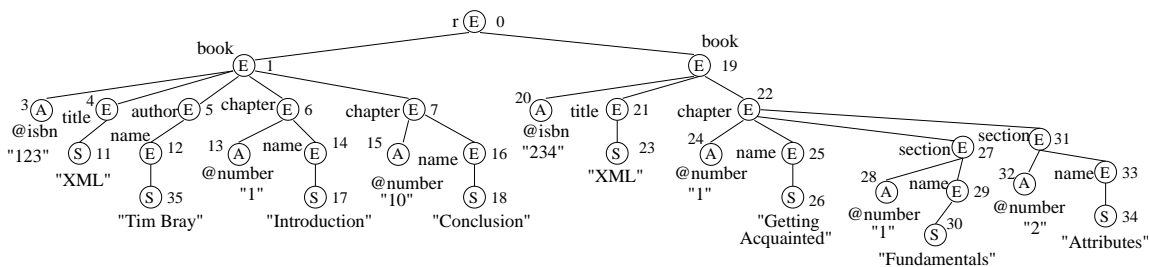


Figure 6.1: Tree representation of XML data

the problem below, using the familiar domain of books.

Example 6.1 Suppose that the XML data (represented as a tree) in Fig. 6.1 is being exchanged and that the initial design of the consumer database has a single table `Chapter` with fields `bookTitle`, `chapterNum` and `chapterName` (written `Chapter(bookTitle, chapterNum, chapterName)`). The table is populated from the XML data as follows: For each `book` element, the value of the `title` subelement is extracted. A tuple is then created in the `Chapter` relation for each `chapter` subelement containing the `title` value for `bookTitle`, the `number` value for `chapterNum`, and the `name` value for `chapterName` (see Fig. 6.2(a) for the resulting relational instance.) The key of the `Chapter` table has been specified as `bookTitle` and `chapterNum`. While importing this XML data, violations of the key are detected because two different books have the same title (“XML”) and disagree on the name of chapter one (“Introduction” versus “Getting Acquainted”). After digging through the documentation accompanying the XML data, the database designers decide to change the schema to `Chapter(isbn, chapterNum, chapterName)` with a key of `isbn` and `chapterNum` (populated in the obvious way from the XML data). The resulting relational instance is shown in Fig. 6.2(b). While importing the XML data, no violations of the key constraint are detected. However, the designers are not sure whether they were lucky with this particular XML data set, or whether such violations will never occur.

It turns out that given the following keys on the XML data, the designers of the consumer database could prove that the key of `Chapter` in their modified design is correct:

1. $(\epsilon, (//book, \{ @isbn \}))$: `isbn` uniquely identifies a `book` element.
2. $(//book, (chapter, \{ @number \}))$: within each `book`, `number` is a key for `chapter`, i.e.,

bookTitle	chapterNum	chapterName
XML	1	Introduction
XML	10	Conclusion
XML	1	Getting Acquainted

(a) Chapter: the initial design

isbn	chapterNum	chapterName
123	1	Introduction
123	10	Conclusion
234	1	Getting Acquainted

(b) Chapter: a refined design

Figure 6.2: Sample relational instances

`number` is a key for `chapter` *relative to* `book`.

3. $(//book, (title, \{\}))$, and $(//book/chapter, (name, \{\}))$: each `book` has a unique `title`, and within each `book`, each `chapter` has a unique `name`.

That is, if these XML keys hold on the data being imported, then $isbn, chapterNum \rightarrow chapterName$ is a functional dependency (FD) that is guaranteed to hold on the `Chapter` relation generated (in other words, $(isbn, chapterNum)$ is a key of the relation). We refer to the FD as one that is *propagated* from these XML keys.

In general, given a transformation to a predefined relational schema and a set Σ of XML keys, one wants to know whether or not an FD is propagated from Σ via the transformation. Let us refer to this problem as *XML key propagation*. The ability to compute XML key propagation is important in checking the consistency of a predefined relational schema for storing XML data. \square

On the other hand, suppose that the relational database is designed from scratch or can be re-designed to fit the constraints (and thus preserve the semantics) of the data being imported. A common approach to designing a relational database is to start with a rough schema (possibly the universal relation) and refine it into a normal form (such as BCNF or 3NF [AHV95]) using FDs. In our scenario, we assume that the designer specifies the rough

schema by a mapping from the XML document. The FDs over that rough schema must then be inferred from the keys of the XML document using the mapping. However, it is impractical to compute the set F of all the FDs propagated since F is exponentially large in the number of attributes. We would therefore like to find a *minimum cover* [AHV95] of F , that is, a subset F_m of F that is equivalent to F (i.e., all the FDs of F can be derived from F_m using Armstrong's Axioms, which are well-known for inferring relational FDs) and is non-redundant (i.e., none of the FDs in F_m can be derived from other FDs in F_m).

Example 6.2 Returning to our example, suppose that the database designers decide to start from scratch and initially propose a schema of `Chapter(isbn, booktitle, author, chapterNum, chapterName)`, with the obvious mapping from the data in Fig. 6.1. From the four keys given earlier, the following minimum cover for `Chapter` can be derived:

$$\text{isbn} \rightarrow \text{bookTitle}, \quad \text{isbn, chapterNum} \rightarrow \text{chapterName}.$$

Using these dependencies, the following BCNF decomposition of the initial design would be produced: `Book(isbn, bookTitle)`, `Author(isbn, author)` and `Chapter(isbn, chapterNum, chapterName)`. Note that `isbn` \rightarrow `author` is not mapped from the keys since a book may have several authors. \square

In this chapter we present a framework for improving consumer relational database design. Our approach is based on inferring functional dependencies from XML keys through a given mapping (transformation) of XML data to relations. The class of XML keys considered is the class of strong keys, where key paths are simple attributes. The class, denoted as \mathcal{K}_{att} , is defined on Section 4.2.2 of Chapter 4. We believe that the keys considered represent an important class commonly found in practice, e.g., in bioinformatic applications, and is a subset of those in XML Schema [Tho02].

Although a number of relational storage techniques have been developed for XML [STZ⁺99, Sha01, Ora01, SKWW00, MFK⁺00, LC01], to the best of our knowledge, our framework and algorithms are the first results on mapping XML constraints through relational views. Being able to reason about constraints on views not only plays an important role in the design of relational storage of XML data, but it is also useful for query optimization and

data integration.

The rest of the chapter is organized as follows. Section 6.1 defines our transformation language. The constraint propagation problem is formally defined in Section 6.2, along with undecidability results that give practical motivation for the restrictions adopted in the framework. In particular, we show that it is impossible to derive relational dependencies from XML constraints if either a transformation language is too rich or the XML constraints include foreign keys. The algorithms for key propagation are presented in Sections 6.3 and 6.4. First, we present a polynomial time algorithm for checking whether an FD on a predefined relational database is propagated from a set of XML keys via a transformation. Then, in Section 6.4, we present a polynomial-time algorithm that, given a universal relation specified by a transformation rule and a set of XML keys, finds a minimum cover for all the functional dependencies mapped from XML keys. Experimental results follow in Section 6.5. They show that the algorithms are efficient in practice. Section 6.6 describes related work, and possible extensions of the framework.

Note that the polynomial-time algorithm for finding a minimal cover from a set of XML keys is rather surprising, since it is known that a related problem in the relational context – finding a minimum cover for functional dependencies *embedded* in a subset of a relation schema – is inherently exponential [Got87].

6.1 Transformations from XML to Relations

The transformation language given below is quite simple, and forms a core of many common transformations found throughout the literature, in particular those of [STZ⁺99].

Definition 6.1 *A transformation σ from XML to relations of schema $\mathbf{R} = (R_1, \dots, R_n)$ is specified as $(\text{Rule}(R_1), \dots, \text{Rule}(R_n))$, where each $\text{Rule}(R_i)$, referred to as the table rule for R_i , is defined with:*

- a set X_i of variables, in which x_r is a distinguished variable, referred to as the root variable;

- a set of field rules $\{l : \text{value}(x) \mid l \in \text{att}(R_i)\}$, where x is a distinct variable in X_i , and $\text{att}(R_i)$ denotes the set of attributes in the schema of relation R_i ;
- a set of variable mapping rules of the form $x \leftarrow y/P$, where $x, y \in X_i$ and P is a path expression.

In addition, each variable $x \in X_i$ is connected to the root variable x_r ; that is, x is specified with either $x \leftarrow x_r/P$ in the rule, or $x \leftarrow y/P$ and y is connected to x_r ; moreover, for any $x \leftarrow y/P$, 1) P is a simple path (i.e. without $//$) unless y is x_r , and 2) no field rule is defined as $l : \text{value}(y)$ when there exists a variable x specified with $x \leftarrow y/P$, that is, y does not indicate a text node.

Intuitively, σ defines a set of table rules, one for each relation R_i . Given an XML tree T , $\text{Rule}(R_i)$ generates an instance of R_i from T . Specifically, $\text{Rule}(R_i)$ is defined with a set X_i of variables that range over nodes in T , along with a set of field rules which populate the fields (columns) of the table with the values of these nodes. A variable rule $x \leftarrow y/P$ indicates that if the variable y denotes a node in T , then the variable x ranges over nodes in $y[[P]]$. A field rule $l : \text{value}(x)$ computes the serialized value of the subtree rooted at x (to be explained later) for the l field. The condition in the definition is to ensure that all mapping rules make sense, i.e. there exists some XML tree T such that for any $x \leftarrow y/P$ in the rule, $y[[P]]$ is indeed a set of nodes reachable from the root of T .

Example 6.3 Expanding on Example 6, consider the following schema \mathbf{R} (with keys underlined):

```

book(isbn, title, author, contact)
chapter(inBook, number, name)
section(inChapt, number, name)

```

A transformation σ from the XML data of Fig. 6.1 to \mathbf{R} could be specified as:

$$\sigma = (\text{Rule}(\text{book}), \text{Rule}(\text{chapter}), \text{Rule}(\text{section}))$$

$$\text{Rule}(\text{book}) = \{ \text{isbn}: \text{value}(x_1), \text{title}: \text{value}(x_2), \text{author}: \text{value}(x_3), \text{contact}: \text{value}(x_4) \},$$

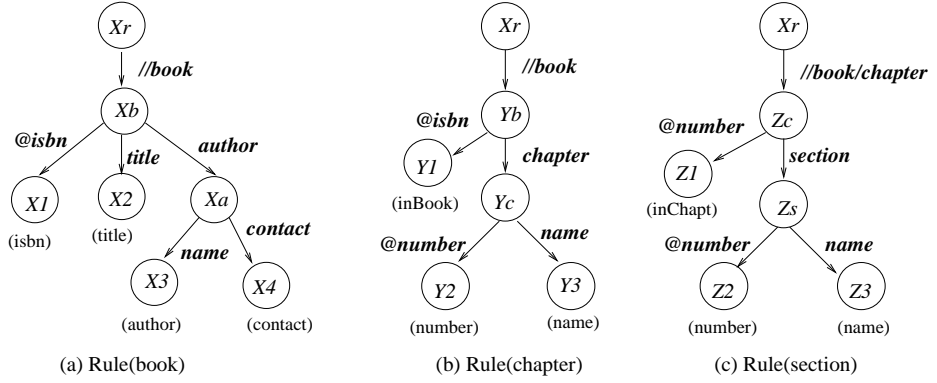


Figure 6.3: Table trees

$$\begin{aligned}
 x_b &\leftarrow x_r//\text{book}, & x_1 &\leftarrow x_b/@\text{isbn}, & x_2 &\leftarrow x_b/\text{title}, \\
 x_a &\leftarrow x_b/\text{author}, & x_3 &\leftarrow x_a/\text{name}, & x_4 &\leftarrow x_a/\text{contact};
 \end{aligned}$$

$$\begin{aligned}
 \text{Rule}(\text{chapter}) &= \{ \text{inBook: value}(y_1), \text{ number: value}(y_2), \text{ name: value}(y_3) \}, \\
 y_b &\leftarrow x_r//\text{book}, & y_1 &\leftarrow y_b/@\text{isbn}, & y_c &\leftarrow y_b/\text{chapter}, \\
 y_2 &\leftarrow y_c/@\text{number}, & y_3 &\leftarrow y_c/\text{name};
 \end{aligned}$$

$$\begin{aligned}
 \text{Rule}(\text{section}) &= \{ \text{inChapt: value}(z_1), \text{ number: value}(z_2), \text{ name: value}(z_3) \}, \\
 z_c &\leftarrow x_r//\text{book}/\text{chapter}, & z_1 &\leftarrow z_c/@\text{number}, & z_s &\leftarrow z_c/\text{section}, \\
 z_2 &\leftarrow z_s/@\text{number}, & z_3 &\leftarrow z_s/\text{name}.
 \end{aligned}$$

□

Table trees. Throughout the remainder of the chapter, we will use an abstract representation of a table rule called a *table tree*. In a table tree T_R representing $\text{Rule}(R)$, each variable in $\text{Rule}(R)$ corresponds to a unique node, and edges are labeled with path expressions. For example, Figure 6.3 depicts table trees for the table rules given in Example 6.1.

Semantics. Given an XML tree T , each $\text{Rule}(R_i)$ maps T to an instance I_i of R_i . More specifically, given a variable specification $x \leftarrow y/P$, x ranges over $y[[P]]$; x_r is always interpreted as the root r . A field rule $l : \text{value}(x)$ populates the l field with values in $\{\text{value}(x) \mid x \in y[[P]]\}$, where the function value in a field rule returns a string representing the pre-order traversal of the subtree rooted at x . Let $\text{att}(R_i) = \{l_1, \dots, l_k\}$ and each variable x be specified with $x \leftarrow x'/P_x$. Then the instance I_i is generated by $I_i = \{(l_1 : \text{value}(x_1), \dots, l_k : \text{value}(x_k)) \mid x_r = r, x \in x'[[P_x]], x \in X_i\}$.

book				chapter		
isbn	title	author	contact	inBook	number	name
123	XML	Tim Bray	null	123	1	Introduction
234	XML	null	null	123	10	Conclusions
				234	1	Getting Acquainted

section		
inChapt	number	name
1	1	Fundamentals
1	2	Attributes

Figure 6.4: Instances generated by the transformation of Example 6.1

Example 6.4 Rule(*section*) is interpreted as:

$$\{(inChapt : value(z_1), number : value(z_2), name : value(z_3)) \mid \\ z_c \in r[//book/chapter], z_1 \in z_c[@number], z_s \in z_c[section], \\ z_2 \in z_s[@number], z_3 \in z_s[name]\}.$$

Referring to the XML tree T in Fig. 6.1, $value(6) = (name : (S : Introduction), @number : 1)$. The interpretation of the table rules of Example 6.1 over T generates the relational instance of Fig. 6.4. \square

Several subtleties are worth mentioning. First, since XML data is semistructured in nature it is likely that for $x \leftarrow y/P$, $y[P]$ is empty. In this case $value(x)$ is defined to be `null`. Second, if $y[P]$ has multiple elements, then to generate the relation with the table rule containing x , an implicit Cartesian product is computed so that all nodes in $y[P]$ are covered in the relation. That is, for each $x \in y[P]$, there is a tuple t whose l field has $value(x)$. Finally, note that the transformation language is capable of expressing projection (π), Cartesian product (\times) and set union (\cup ; limited in the sense that it is implemented with “//”). We shall compare our approach with others published in the literature [STZ⁺99, BFRS02, MFK01, SKWW00] in Section 6.6.

6.2 Problem Statement and Limitations

Having described our transformations, we next state the central technical problems investigated in this chapter: the key propagation problem and the problem of finding a minimum cover.

6.2.1 Key propagation

The question of *key propagation* asks if given a transformation σ from XML data to relations of a fixed schema \mathbf{R} and an XML tree T satisfying a set Σ of XML keys, whether $\sigma(T)$ satisfies an FD φ (on a relation schema R in \mathbf{R}). We write $\Sigma \models_{\sigma} R : \varphi$ if the implication holds for all XML trees satisfying Σ , and refer to φ as an FD *propagated from* Σ . With respect to a transformation specification language, the *key propagation problem* is to determine, given any σ expressed in the language, any XML keys Σ and an FD φ , whether or not $\Sigma \models_{\sigma} R : \varphi$. Note that we do not require the XML data to conform to any DTD or other type specification.

A subtle issue arises from `null` values in $\sigma(T)$, the relations generated from an XML tree T via σ . In particular, there may exist R tuples in $\sigma(T)$ with FD $X \rightarrow Y$ such that their X or Y fields contain `null`. The presence of `null` complicates FD checking since comparisons of `null` with any value do not evaluate to a Boolean value (`true` or `false` [RG00]). A brutal solution is to restrict the semantics of the transformation σ so that a tuple is not included if it has a `null` field. Since XML is semistructured, this could exclude a large number of “incomplete” tuples from $\sigma(T)$. We therefore adopt the following semantics of FDs: $\sigma(T)$ satisfies the FD $X \rightarrow Y$, denoted by $\sigma(T) \models X \rightarrow Y$, iff

- for any tuple t in R , if $\pi_X(t)$ contains `null` then so does $\pi_Y(t)$;
- for tuples t_1, t_2 in R , if neither $\pi_{XY}(t_1)$ nor $\pi_{XY}(t_2)$ contains `null` and $\pi_X(t_1) = \pi_X(t_2)$, then $\pi_Y(t_1) = \pi_Y(t_2)$.

The motivation behind the first condition is that an FD is possibly treated as a key when normalizing the relational schema, and an “incomplete key” X cannot determine complete

Y fields. We call this condition as the *null restriction* for FD satisfaction. Observe that under the new semantics, the reflexivity rule of the Armstrong Axioms for FD inference has to be adjusted. This is because, given any set of labels X , there is no guarantee that for any subset Y of X , $X \rightarrow Y$; indeed, a tuple t may contain `null` in fields of $X \setminus Y$, but not in fields of Y . Thus, an FD is only considered trivial if it is of the form $X \rightarrow X$, and the reflexivity rule is now stated as: for any set of labels X , $X \rightarrow X$. Note that the augmentation and transitivity rules of the Armstrong Axioms remain unchanged. One consequence of this new set of rules is that although the union rule can be inferred from the new rules, the decomposition rule cannot. That is, if $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$. But if $X \rightarrow YZ$ we cannot infer that $X \rightarrow Y$. This is because although it is known that whenever X contains `null` so does YZ , we are not sure that Y alone also contains `null`.

Example 6.5 Let σ be the transformation $\text{Rule}(\textit{book})$ given in Example 6.1 and Σ be the set of keys defined in the introduction. Then $\Sigma \models_{\sigma} \textit{book: isbn} \rightarrow \textit{title}$ since in any XML tree T satisfying Σ : (1) an `@isbn` attribute uniquely determines a *book* element; and (2) below each *book* element there is at most one node reachable via path *title*. Thus given the definition of $\text{Rule}(\textit{book})$, among complete *book* tuples (in which *title* is not `null`), an *isbn* field uniquely determines a *title* field. \square

It has been shown that the implication problem of full dependencies on views, which are defined with SPCU (selection, projection, cartesian product, and union) queries, is decidable both in the relational model [AHV95] and in the nested relational model [PT99]. Our transformation language is very simple, and can only express projection (π), Cartesian product (\times) and a limited form of set union (\cup). Based on the previous decidability results, it is likely that the language can be extended to express selection and a general form of set union without significantly affecting our framework for key propagation. But one might be tempted to develop a even richer language, which can express all relational algebra operators, including set difference ($-$). Although these operators can be generalized to XML trees (e.g., with our *value* function), the following negative result holds:

Theorem 6.1 *The key propagation problem from XML to relational data is undecidable when the transformation specification language can express all relational algebra operators.*

Proof. We prove the theorem by reduction from the equivalence problem for relational algebra queries: given any relation schema S, R and relational algebra queries Q_1, Q_2 from S to R , determine whether for any instance I of S , $Q_1(I) = Q_2(I)$ (denoted by $Q_1 \equiv Q_2$). This is a well-known undecidable problem [Var81].

We give the reduction as follows. Let R' be an extension of R with the addition of a new field $@l$, i.e., $Attr(R') = Attr(R) \cup \{@l\}$ where $@l$ is not in $Attr(R)$ and $Attr(S)$. We define a mapping σ from XML data to relational databases of schema $\mathbf{R} = \{R'\}$, a set Σ of XML keys, and an FD φ on R' , such that $\Sigma \models_{\sigma} R' : \varphi$ iff $Q_1 \equiv Q_2$. More specifically, let Σ consist of a single absolute key: $(S/t, Attr(S))$. That is, in an XML tree T satisfying Σ , the values of $Attr(S)$ attributes of any S/t node uniquely determine the node. We define the FD φ to be $\emptyset \rightarrow R'$, i.e., there is a unique tuple in the R' relation. Finally, we define the mapping σ to be

$$\{t\} \cup \text{"2"} \times ((Q_1(\pi_{Attr(S)}(S/t)) - Q_2(\pi_{Attr(S)}(S/t))) \cup (Q_2(\pi_{Attr(S)}(S/t)) - Q_1(\pi_{Attr(S)}(S/t))),$$

where t is a R' tuple such that all of its fields have constant value "1", while in all the other tuples the $@l$ fields have a constant value "2"; and $\pi_{Attr(S)}(S/t)$ denotes the values of $Attr(S)$ attributes of S/t nodes.

There is an one-to-one mapping between the set of all instances of S and the $\pi_{Attr(S)}(S/t)$ values of all XML trees satisfying Σ . Thus $E = (Q_1(\pi_{Attr(S)}(S/t)) - Q_2(\pi_{Attr(S)}(S/t))) \cup (Q_2(\pi_{Attr(S)}(S/t)) - Q_1(\pi_{Attr(S)}(S/t)))$ is equal to the empty set \emptyset for all XML trees satisfying Σ iff for all instances I of S , the difference between $Q_1(I)$ and $Q_2(I)$ is \emptyset , i.e., when $Q_1 \equiv Q_2$. Note that $\sigma(T) \models \varphi$ iff $\sigma(T)$ consists of a single tuple. However the tuple t is already in $\sigma(T)$, which is distinguished from the others in $\sigma(T)$ by the value of its $@l$ field. Hence for each XML tree T satisfying Σ , $\sigma(T) \models \varphi$ iff the set E is \emptyset . Thus $\Sigma \models_{\sigma} R' : \varphi$ iff $Q_1 \equiv Q_2$. Hence the encoding is indeed a reduction from the equivalence problem for relational algebra queries to the key constraint propagation problem. Since the former is undecidable, so is the latter. \square

In contrast, as will be seen shortly, for our transformation language there is a polynomial time algorithm in the size of Σ and σ .

The key propagation problem can be similarly defined for XML to XML transformations, and is equally important in that context. It is worth mentioning that popular query languages for XML, such as XQuery [Cha01], subsume our specification language and in fact, when being (naturally) used to specify XML to XML transformations, are undecidable extensions as well.

6.2.2 Minimum cover

The problem of *finding a minimum cover* is to compute, given a universal relation \mathbf{U} and a set Σ of XML keys, a minimum cover F_m for the set F^+ of all FDs on \mathbf{U} propagated from Σ . Guided by F_m , one can then decompose \mathbf{U} into a normal form as illustrated by Example 6. This is analogous to techniques for designing relational databases [AHV95]. In our context, a universal relation is simply the collection of all the fields of interest, along with a table rule that defines these fields.

Example 6.6 Recall the schema \mathbf{R} and the transformation given in Example 6.1. A universal relation \mathbf{U} here is the collection of all the fields of \mathbf{R} , defined as follows:

$\mathbf{U} = (\text{bookIsbn}, \text{bookTitle}, \text{bookAuthor}, \text{authContact}, \text{chapNum}, \text{chapName},$
 $\text{secNum}, \text{secName}),$

$\text{Rule}(\mathbf{U}) = \{\text{bookIsbn: value}(x_1), \text{bookTitle: value}(x_2), \text{bookAuthor: value}(x_3),$
 $\text{authContact: value}(x_4), \text{chapNum: value}(y_1), \text{chapName: value}(y_2),$
 $\text{secNum: value}(z_1), \text{secName: value}(z_2)\},$

$x_b \leftarrow x_r // \text{book}, x_1 \leftarrow x_b / @\text{isbn}, x_2 \leftarrow x_b / \text{title}, x_a \leftarrow x_b / \text{author},$

$x_3 \leftarrow x_a / \text{name}, x_4 \leftarrow x_a / \text{contact}, y_c \leftarrow x_b / \text{chapter}, y_1 \leftarrow y_c / @\text{number},$

$y_2 \leftarrow y_c / \text{name}, z_s \leftarrow y_c / \text{section}, z_1 \leftarrow z_s / @\text{number}, z_2 \leftarrow z_s / \text{name}$

The table tree of $\text{Rule}(\mathbf{U})$ is depicted in Fig. 6.5 (a).

Suppose the following set of keys are defined:

- $KS_1 : (\epsilon, (//\text{book}, \{\text{@isbn}\}))$: within the context of the entire document a book element is identified by its @isbn attribute.

- $KS_2 : (//book, (chapter, \{@number\}))$: within the context of any subtree rooted at a book node, a chapter is identified by its $@number$ attribute.
- $KS_3 : (//book, (title, \{\}))$: each book has at most one title; similarly, $KS_4 : (//book/chapter, (name, \{\}))$ for the name of a chapter, and $KS_5 : (//book/chapter/section, (name, \{\}))$ for section name.
- $KS_6 : (//book/chapter, (section, \{@number\}))$: within the context of a chapter of a book, each section is identified by its $@number$ attribute.
- $KS_7 : (//book, (author/contact, \{\}))$: a book can have multiple authors, but at most one has contact information (the contact author).

From this set of XML keys, the following minimum cover for the FDs on \mathbf{U} can be computed:

$$\begin{aligned} \text{bookIsbn} &\rightarrow \text{bookTitle}, & \text{bookIsbn} &\rightarrow \text{authContact}, \\ \text{bookIsbn, chapNum} &\rightarrow \text{chapName}, & \text{bookIsbn, chapNum, secNum} &\rightarrow \text{secName}. \end{aligned}$$

Guided by these FDs, we can decompose \mathbf{U} into BCNF:

$$\begin{aligned} &\text{book}(\underline{\text{bookIsbn}}, \text{bookTitle}, \text{authContact}), \\ &\text{author}(\underline{\text{bookIsbn}}, \text{bookAuthor}), \\ &\text{chapter}(\underline{\text{bookIsbn}}, \underline{\text{chapNum}}, \text{chapName}), \\ &\text{section}(\underline{\text{bookIsbn}}, \underline{\text{chapNum}}, \underline{\text{secNum}}, \text{secName}) \quad \square \end{aligned}$$

Although in the relational context algorithms have been developed for computing a minimum cover for a set of FDs [BB79, Got87, Mai80], they cannot be used in our context since the FDs must be computed from the XML keys Σ via the transformation σ , instead of being provided as input as for those relational algorithms. Furthermore, relational FDs are not capable of expressing XML keys and vice versa.

6.2.3 Propagation of other XML constraints

XML Schema supports both keys and foreign keys. A simple form of *foreign keys* is $(Q_1, [@l_1, \dots, @l_n]) \subseteq (Q_2, [@l'_1, \dots, @l'_n])$, where Q_1, Q_2 are path expressions and $@l_i, @l'_i$

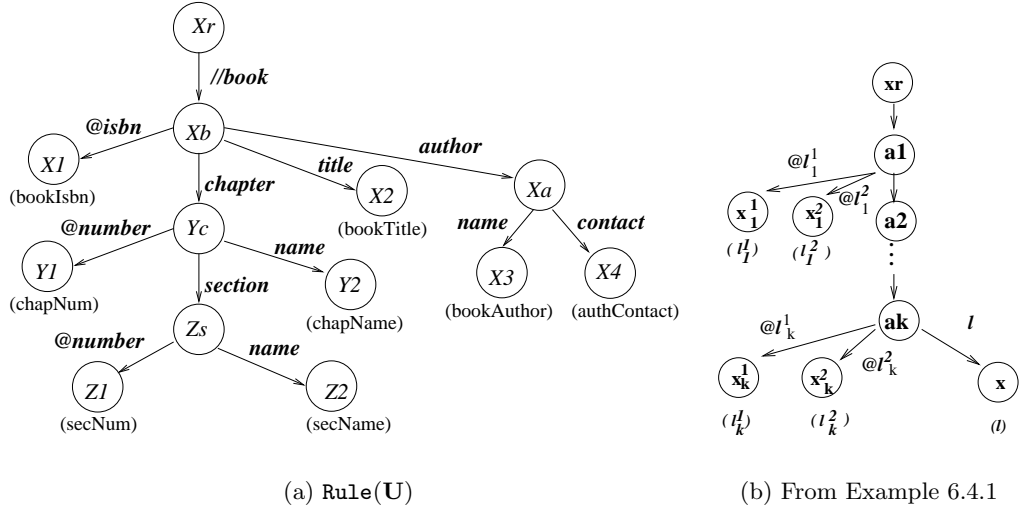


Figure 6.5: Table trees

are attributes. It asserts that for any node n_1 in $\llbracket Q_1 \rrbracket$ there exists n_2 in $\llbracket Q_2 \rrbracket$ such that $n_1.\@l_i = n_2.\@l_i$ for all $i \in [1, n]$, and that $\@l'_1, \dots, \@l'_n$ are a key of nodes in $\llbracket Q_2 \rrbracket$. That is, the list $[\@l_1, \dots, \@l_n]$ is a foreign key of nodes in $\llbracket Q_1 \rrbracket$ referencing nodes in $\llbracket Q_2 \rrbracket$. Although it is tempting to develop algorithms to compute the propagation of both keys and foreign keys, we have the following negative result:

Theorem 6.2 *The propagation problem for XML keys and foreign keys is undecidable for any transformation language that can express identity mapping.*

Proof. We prove the theorem by reduction from the implication problem for keys by keys and foreign keys in the relational model: given a relational schema \mathbf{R} , a key ϕ on relation schema $R \in \mathbf{R}$, and a set of keys and foreign keys Θ over \mathbf{R} , decide whether or not $\Theta \models R : \phi$. This is an undecidable problem [FL02].

We give the reduction by defining an identity mapping σ from XML data to relational databases of schema \mathbf{R} , and a mapping from a set Θ of relational keys and foreign keys to a set Σ of XML keys and foreign keys, such that $\Sigma \models_\sigma R : \phi$ iff $\Theta \models R : \phi$.

The identity mapping is one in which the XML representation of a relational database of \mathbf{R} is mapped to the same database. Intuitively we adopt the canonical XML representation

of relations: the root of the XML tree has a child R for every R in \mathbf{R} ; each R has children t encoding the tuples of an instance of R ; and each t child of R has attributes appearing in a tuple of the instance. In our language this can be expressed as a transformation defined with paths of length 3:

$$\sigma = \{\mathbf{Rule}(R) \mid R \in \mathbf{R}\}$$

$$\mathbf{Rule}(R) = \{l_1 : \text{value}(y_1), \dots, l_n : \text{value}(y_n)\},$$

$$x_t \leftarrow x_r/R/t, \quad y_1 \leftarrow x_t/@l_1, \dots, y_n \leftarrow x_t/@l_n, \quad \text{where } \text{Attr}(R) = \{l_1, \dots, l_n\}$$

We encode relational constraints Θ in terms of XML constraints Σ as follows.

1. for each key: $R[l_1, \dots, l_k] \rightarrow R$, we generate an XML key: $(\epsilon, (R/t, \{\@l_1, \dots, \@l_k\}))$.
2. for each foreign key: $R[l_1, \dots, l_k] \subseteq R'[l'_1, \dots, l'_k]$ and $R'[l'_1, \dots, l'_k] \rightarrow R$, we generate an XML foreign key: $(R/t, \{\@l_1, \dots, \@l_k\}) \subseteq (R'/t', \{\@l'_1, \dots, \@l'_k\})$.

We show that the above encoding is indeed a reduction by proving that $\Theta \models R : \phi$ iff $\Sigma \models_{\sigma} R : \phi$.

First, for any instance I of \mathbf{R} , such that $I \models \bigwedge \Theta \wedge \neg\phi$, we can construct an XML tree T as follows. Each relation R in \mathbf{R} corresponds to an element under the root node, and each tuple t of $\pi_R(I)$ corresponds to an element in the tree reached by following path R/t , and with attributes $\text{Attr}(R)$ populated with values of t . It is easy to see that $T \models \Sigma$. Since σ maps the tree representation of I back to I , $\sigma(T) \not\models \phi$. Therefore, if $\Theta \not\models R : \phi$, then $\Sigma \not\models_{\sigma} R : \phi$.

For the other direction, consider any XML tree T such that $T \models \Sigma$, yet $\sigma(T) \not\models R : \phi$. Since all paths involved in constraints of Σ have a maximum length of three, parts of T with depth greater than three or not mentioned in the constraint's path expressions are irrelevant. Since σ is an identity mapping from the tree representation T of I back to I , if $T \models \Sigma$ then $I \models \Theta$, and $\sigma(T) = I \not\models R : \phi$. Therefore, if $\Sigma \not\models_{\sigma} R : \phi$ then $\Sigma \not\models R : \phi$.

This shows that the above encoding is indeed a reduction from the implication problem of keys by keys and foreign keys in the relational model. \square

Because of this negative result, we restrict our attention to the propagation of XML keys.

6.3 Checking Key Propagation

Checking key propagation is nontrivial for a number of reasons: First, XML data is semistructured in nature, which complicates the analysis of key propagation by the presence of null values. Second, XML keys which are not in Σ but are consequences of Σ may yield FDs on a relational view. Thus key propagation involves XML key implication. Third, XML data is hierarchically structured and thus XML keys are relative in their general form – they hold on a sub-document. However, its relational view collapses the hierarchical structures into a flat table and thus FDs are “absolute” – they hold on the entire relational view. Thus one needs to derive a unique identification of a sub-document from a set of relative keys.

Example 6.7 To illustrate this last point, consider again the transformation σ given in Example 6.1 and the set Σ of XML keys given in Example 6.2.2. From KS_2 , a key for **chapter**, one might expect an FD **number** \rightarrow **name** to hold on the *chapter* relation. This is, however, not the case since $@number$ is a key of **chapter** relative to a particular **book**; thus to uniquely identify a **chapter** in the entire document one also needs a key for **book**. Reflecting this in the relation *chapter*, one can expect that **number, inBook** \rightarrow **name** is an FD, where **inBook** is a key for *book* by KS_1 and σ . However, KS_1 and KS_2 alone do not yield the FD, because XML data is semistructured and thus a **chapter** may have several **name** subelements. One can conclude that it is indeed an FD propagated from Σ only after KS_4 is taken into account, which says each **chapter** has at most one **name**. Putting these together, the FD is derived from a set of XML keys consisting of KS_1, KS_2 and KS_4 . Observe that, however, neither **inChapt, number** \rightarrow **name** is an FD propagated from Σ on the *section* relation, nor is **isbn** \rightarrow **author** an FD on the *book* relation. \square

As illustrated in the example, to uniquely identify a node within the entire document we need a set of XML keys identifying unique contexts up to the root. This notion has been formalized by the definition of a *transitive* set of keys in Chapter 4 (Section 4.1.4). We say

that a node is *keyed* if there exists a transitive set of keys to uniquely identify the node.

Example 6.8 The set $\{KS_1, KS_2\}$ is transitive since any chapter in the document can be identified by providing `@isbn` of a book and `@number` of a chapter. Thus every chapter node is keyed. In contrast, $\{KS_2\}$ is *not* transitive since with it alone there is no way to uniquely identify a book in the document, which is necessary before identifying a chapter of that book. \square

6.3.1 Propagation Algorithm

Before we present the algorithm, we first introduce a few notions.

Table tree. Algorithm `propagation` uses the tree representation of a transformation to bridge the gap between XML keys and the FD ϕ to be checked. Without loss of generality, assume that ϕ is of the form $Y \rightarrow Z$ with $(Y \cup Z) \subseteq \text{Attr}(R)$, and that $\text{Rule}(R)$ is $\{l_i : \text{value}(x_i) \mid i \in [1, m]\}$ along with a set X of variables and mappings $x \leftarrow y/P$ for each $x \in X$. In the table tree T_R representing $\text{Rule}(R)$, any variable x in X has a unique node corresponding to it, referred to as the x -node. In particular, the x_r -node is the root of T_R . Observe that for any $x, y \in X$, if the x -node is a descendant of the y -node in T_R , then there is a unique path in T_R from the y -node to the x -node, which is a path expression. We denote the path by $P(y, x)$, which exists only if there are variables x_1, \dots, x_k in X such that $x_1 = y$, $x_k = x$ and for each $i \in [1, k - 1]$, $x_{i+1} \leftarrow x_i/P_i$ is a mapping in $\text{Rule}(R)$. We use $\text{descendants}(y)$ to denote the set of all the variables that are descendants of y ; we define $\text{ancestors}(y)$ similarly. In particular, if x is specified with $x \leftarrow y/P$ then the variable y is called the *parent* of x , denoted by $\text{parent}(x)$, and $x \in \text{children}(y)$. Referring to Figure 6.3 (c), for example, x_r is the parent of Z_c , and $P(x_r, Z_c)$ is `//book/chapter`.

Algorithm. The intuition behind Algorithm `propagation` is as follows. Given an FD $\phi = Y \rightarrow Z$ on R , assume that the table tree representing $\text{Rule}(R)$ is T_R , and that each $l_i \in Z$ is specified with $\text{value}(x_i)$. Let $\text{var}(Z)$ be the set of variables that populate fields in Z ; that is, $\text{var}(Z) = \{x \mid l : \text{value}(x) \in \text{Rule}(R), l \in Z\}$. Then $\Sigma \models_\sigma Y \rightarrow Z$ iff (1) for every $l_i \in Z$, either $l_i \in Y$, or x_i is the root node, or there exists a variable *target*

in T_R such that *target* is an ancestor of some $z \in \text{var}(Z)$, and is keyed with fields of Y . Moreover, either x_i is *unique* under *target* or x_i is *unique* under some ancestor of *target*. In other words, there is a set of transitive keys that uniquely identifies *target* with only those attributes which define fields of Y , $\Sigma \models (P(x_r, \text{target}), (P(\text{target}, z), \{\}))$, and $\Sigma \models (P(x_r, x'), (P(x', x_i), \{\}))$, where x' is *target* or some ancestor of *target*; (2) every field of Y is defined by an attribute of some ancestor of an x_i that is required to exist.

The first condition asserts that for any R tuples t_1 and t_2 , if they agree on their Y fields and do not contain **null**, then they agree on their Z fields. The second condition excludes the possibility that in some R tuple t , fields in Z are defined while some of their Y fields are **null**.

Example 6.9 Consider the transformation rule below, depicted as a table tree in Figure 6.6(a).

$$\begin{aligned} \text{Rule}(R) = \{ & l_1: \text{value}(x_1), \quad l_{y_1}: \text{value}(y_1), \quad l_{y_2}: \text{value}(y_2), \quad l_z: \text{value}(z)\}, \\ & x' \leftarrow x_r/Q_1, \quad \text{target} \leftarrow x'/Q_2, \quad y \leftarrow \text{target}/Q_3, \quad z \leftarrow y/Q_4, \\ & y_1 \leftarrow \text{target}/@l_1, \quad y_2 \leftarrow y/@l_2, \quad x_1 \leftarrow x'/Q_5 \end{aligned}$$

From the following set Σ of XML keys:

$$\begin{aligned} \phi_1 : (Q_1/Q_2, \{@l_1\}) & \quad \phi_2 : (Q_1/Q_2, (Q_3/Q_4, \{\})) \\ \phi_3 : (Q_1, (Q_5, \{\})) & \quad \phi_4 : (Q_1/Q_2/Q_3, (\epsilon, \{@l_2\})) \end{aligned}$$

we can conclude that $\Sigma \models_{\sigma} l_{y_1} \rightarrow l_z$ by only considering keys ϕ_1 and ϕ_2 , which assert that the *target*-node is keyed with $@l_1$, and that z is unique under *target*, respectively. By also considering ϕ_3 , we can conclude that $\Sigma \models_{\sigma} l_{y_1} \rightarrow l_z, l_1$. This is because a given value of $@l_1$ uniquely identifies a node in $\llbracket Q_1/Q_2 \rrbracket$, and along this path there exists at most one x_1 -node that populates l_1 . However, it is *not* the case that $\Sigma \models_{\sigma} l_{y_1} \rightarrow l_1$, because this FD violates the null restriction for FD satisfaction. That is, it is *not* the case that whenever in a tuple t , $t.l_{y_1} = \text{null}$ then $t.l_1 = \text{null}$, since the existence of an x_1 -node does not guarantee the existence of a y_1 -node. On the other hand, if $@l_2$ is a required attribute for y , as defined by ϕ_4 , $\Sigma \models_{\sigma} l_{y_1}, l_{y_2} \rightarrow l_z$, and $\Sigma \models_{\sigma} l_{y_1}, l_{y_2} \rightarrow l_z, l_1$. \square

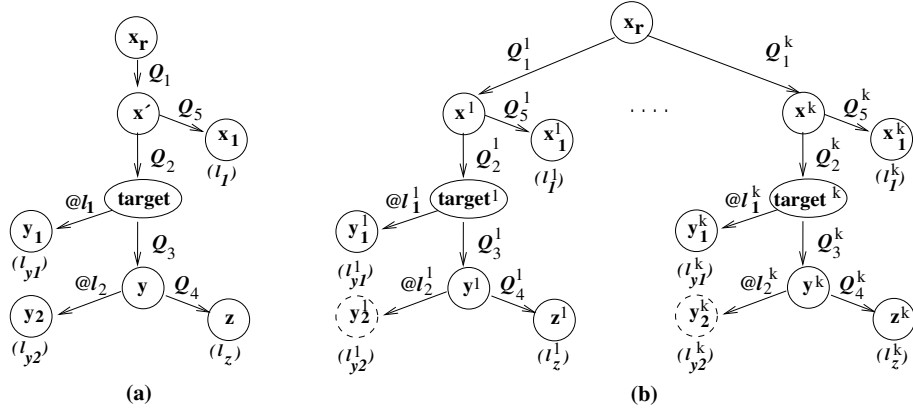


Figure 6.6: Table trees to illustrate key propagation

To keep track of the information needed to check FD propagation we associate the following with each variable x in $\text{Rule}(R)$:

- $att[x]$: the set of attributes of the x -node that populate fields in $Attr(R)$, and are required to exist for x .
- $required[x]$: the set of fields populated with required attributes of the x -node.
- $unique[x]$: the set of fields populated with unique descendants of the x -node.
- $ancestor[x]$: the list of ancestors of x , starting from the root x_r .

Note that if $y \leftarrow x/@a$ is a variable mapping, $l : value(y) \in \text{Rule}(R)$, and there exists a key $(Q, (Q', S)) \in \Sigma$ such that $@a \in S$ then $@a \in att[x]$, and $l \in required[x]$. Moreover, $l \in unique[x]$ since any node in an XML tree has at most one attribute labeled with a particular name. Therefore, $required[x] \subseteq unique[x]$.

Putting everything together, Algorithm `propagation` is shown in Figure 6.7. The algorithm first verifies if the FD to be checked is trivial, that is, if $Y = Z$, and returns *true* if this is the case. Then, for each variable $x \in X$ it computes the associated information $att[x]$, $unique[x]$, $required[x]$, and $ancestor[x]$ by invoking procedure `initializeVars` (Line 2). The algorithm uses *Ycheck* and *Zcheck* to verify if the two conditions defined for FDs are satisfied. That is, *Ycheck* is used to ensure that whenever fields in Z are defined then fields in Y are also defined; that is, they do not contain `null`. This is the case if all fields in Y are defined with attributes that are required either for variables that populate

fields in Z or for their ancestors. Therefore, in the algorithm, the initial value of $Ycheck$ is set to $(Y \setminus required[x_r])$ (Line 3) because every required attribute of the root node is present in any XML tree. Then, fields that are required either for variables in $var(Z)$ or their ancestors are removed from the set (Lines 4 to 8). If at the end of this process, $Ycheck$ becomes empty, we can conclude that the condition is satisfied.

The algorithm uses $Zcheck$ to verify the other condition for FD satisfaction. That is, if tuples that agree on their Y fields also agree on their Z fields. This is the case if every field in Z is either unique under a keyed ancestor $target$, or unique under an ancestor of $target$. The initial value of $Zcheck$ is set to $(Z \setminus unique[x_r])$ (Line 9) because unique fields of the root are constants in any relation. The algorithm proceeds by considering each field $l \in Zcheck$, where l is specified with $l : value(x)$. First, the algorithm removes l from $Zcheck$ if $l \in Y$ or if x is the root variable (Line 11). It then traverses the table-tree T_R top-down along the ancestor path from the root x_r to x (Lines 12 to 25), and for each ancestor $target$ in this path, checks if $target$ is keyed (Line 17). The central part of the algorithm is to check whether there is a set of transitive keys for $target$. To do so, it uses variable $context$ to keep track of the closest ancestor for which a key has been found. That is, $target$ is keyed iff $\Sigma \models (P(x_r, context), (P(context, target), att[target]))$, i.e., the attributes of $target$ are a key of $target$ relative to its closest ancestor with a key. XML key implication is checked by invoking Algorithm `implication` given in Figure 5.6 of Chapter 5. If it holds, the algorithm moves $context$ down to $target$ (Line 18); then, it sets a Boolean flag $isKeyed$ to `true` if x is unique under $target$ (Line 19). Moreover, it removes from $Zcheck$ every unique field along the path from $target$ to the root x_r . The algorithm returns `true` iff $Zcheck$ and $Ycheck$ becomes empty, i.e., the two conditions given above are satisfied.

Example 6.10 To illustrate the algorithm, recall the transformation σ of Example 6.1 and the set Σ of XML keys of Example 6.2.2. Consider FD: `isbn` \rightarrow `contact` over relation `book` defined by `Rule(book)`, which is depicted in Figure 6.3 (a). Note that the field `contact` in the FD is specified with variable x_4 . Given Σ , σ and the FD, the algorithm starts the traversal on the ancestors of x_4 , which consists of x_r , x_b (resp. `//book`) and x_a (resp. `//book/author`). First, it checks if x_r is keyed by inspecting $\Sigma \models (\epsilon, \{\})$. Since this

Algorithm propagation

Input: XML keys Σ , FD $\phi = Y \rightarrow Z$ over R , and $\text{Rule}(R)$ in a transformation σ .

Output: true iff $\Sigma \models_{\sigma} R : \phi$.

1. if $Y = Z$ then return *true*;
2. **initializeVars**(x_r);
3. $Ycheck := Y \setminus \text{required}[x_r]$;
4. for each $l \in Z$, where $l : \text{value}(x)$, do
5. $w := x$;
6. while $w \neq x_r$ do
7. $Ycheck := Ycheck \setminus \text{required}[w]$;
8. $w := \text{parent}(w)$;
9. $Zcheck := Z \setminus \text{unique}[x_r]$;
10. for each $l \in Zcheck$, where $l : \text{value}(x)$, do
11. if $l \in Y$ or $x = x_r$ then $Zcheck := Zcheck \setminus \{l\}$;
12. $\text{traverse}[x] := \text{ancestor}[x]$;
13. $\text{context} := x_r$;
14. $\text{isKeyed} := \text{false}$;
15. while not isKeyed and $\text{traverse}[x] \neq \text{nil}$ do
16. $\text{target} := \text{head}(\text{traverse}[x])$;
17. if **implication**($\Sigma, (P(x_r, \text{context}), (P(\text{context}, \text{target}), \text{att}[\text{target}]))$) then
18. $\text{context} := \text{target}$;
19. if $l \in \text{unique}[\text{target}]$ then
20. $\text{isKeyed} := \text{true}$;
21. $w := \text{target}$;
22. while $w \neq x_r$ do
23. $Zcheck := Zcheck \setminus \text{unique}[w]$;
24. $w := \text{parent}(w)$;
25. $\text{traverse}[x] := \text{tail}(\text{traverse}[x])$;
26. return ($Zcheck = \{\}$) and ($Ycheck = \{\}$);

procedure initializeVars (x)

Input: x : a variable in $\text{Rule}(R)$.

Output: $\text{att}[x]$, $\text{required}[x]$, $\text{unique}[x]$, and $\text{ancestor}[x]$ computed.

1. if $x = x_r$ then $\text{ancestor}[x] := \text{nil}$;
2. else $w := \text{parent}(x)$; $\text{ancestor}[x] := \text{ancestor}[w] + (w :: \text{nil})$;
3. $\text{att}[x] := \{\}$; $\text{required}[x] := \{\}$; $\text{unique}[x] := \{\}$;
4. for each y in $\text{descendant}(x)$ do
5. if **implication**($\Sigma, (P(x_r, x), (P(x, y), \{\}))$) then
6. if $l : \text{value}(y)$ is in $\text{Rule}(R)$ then
7. $\text{unique}[x] := \text{unique}[x] \cup \{l\}$;
8. if $y \leftarrow x/@l'$ is a variable mapping and there exists $(Q, (Q', S))$ in Σ
 such that $P(x_r, x) \subseteq Q/Q'$ and $@l' \in S$ then
9. $\text{att}[x] := \text{att}[x] \cup \{@l'\}$;
10. $\text{required}[x] := \text{required}[x] \cup \{l\}$;
11. for each y in $\text{children}(x)$ do **initializeVars**(y);

Figure 6.7: An algorithm for checking XML key propagation

holds by the *epsilon* rule, the algorithm then checks whether x_b is keyed by inspecting $\Sigma \models (//book, \{@isbn\})$. Since this is true, the algorithm proceeds to check whether x_4 is unique under x_b , i.e., whether $\Sigma \models (//book, (author/contact, \{\}))$. This is also the case. In addition, the field `isbn` in the FD is defined in terms of an attribute of x_b , that is required to exist. That is, by the semantics of keys, $(//book, \{@isbn\})$ requires every *book* element to have an `@isbn` attribute. Thus the algorithm concludes that the FD is derived from Σ via σ and returns **true**.

Next, let us consider `Rule(chapter)` of Example 6.1, represented by the table tree of Figure 6.3 (b), and let ϕ be an FD: `inBook,number` \rightarrow `name` over *chapter*. Here the field `name` is defined by the variable y_3 . Given Σ , ϕ and `Rule(chapter)`, Algorithm `propagation` starts the traversal on ancestors of y_3 : x_r, y_b (resp. `//book`) and y_c (resp. `//book/chapter`). After successfully checking that x_r is keyed, it then checks whether y_b is keyed by checking $\Sigma \models (//book, \{@isbn\})$. This is indeed the case; however, y_3 is not unique under y_b . It therefore proceeds to look for a key for y_c relative to y_b , by checking whether $\Sigma \models (//book, (chapter, \{@number\}))$. Again this is the case; furthermore, y_3 is unique under y_c since $\Sigma \models (//book/chapter, (name, \{\}))$. In addition, all the attributes involved in the keys are those that define the fields of the ancestors of y_3 . Thus the algorithm concludes $\Sigma \models_{\sigma} \text{chapter} : \phi$ and returns **true**.

In contrast, the algorithm returns **false** when checking `inChapt,number` \rightarrow `name` over relation *section*, w.r.t. the table rule of Figure 6.3 (c). Here the field `name` is specified with z_3 . After successfully verifying that x_r is keyed, the algorithm checks whether its next ancestor is keyed, i.e., whether $\Sigma \models (//book/chapter, \{@number\})$. This fails. Thus it attempts to verify $\Sigma \models (//book/chapter/section, \{@number\})$, another key relative to the root, which fails again. At this point the algorithm concludes that the FD cannot be derived from Σ and returns **false**. \square

The complexity of the algorithm is $O(m^2n^4)$, where m and n are the sizes of XML keys Σ and table tree T_R , respectively. First, let us analyze procedure `initializeVars`. The cost of function `implication` (Line 5) is $O(m^2n^2)$ since the size of the key to be checked is at most the size of the transformation. Lines 8 to 10 take at most $O(m^2n)$ time. Since each

variable has at most n descendants, the total cost of the procedure is $O(m^2n^3)$. From the main algorithm, procedure `initializeVars` is executed once for each variable in X and therefore the cost of Line 2 of algorithm `propagation` is $O(m^2n^4)$. The computation of `Ycheck` on Lines 3 to 8 cost $O(n^2)$ because the size of Z is $O(n)$, and each variable has at most n ancestors. For computing `Zcheck`, the cost of function `implication` is $O(m^2n^2)$, and it is called at most n^2 times, since the size of `Zcheck` is $O(n)$, each with at most $O(n)$ ancestors. Lines 22 to 24 are executed at most once for each variable in `Zcheck` and cost $O(n)$ time. Therefore, the cost of Lines 9 to 25, and the whole algorithm is $O(m^2n^4)$.

Observe that checking key propagation is related to the problem of mapping constraints through views, which has been well-studied for relational data [Klu80, KP82, MMS79, BV84b]. But the results established there cannot be applied in our context since the transformation language differs, and XML keys cannot be captured by FDs, and vice-versa. Moreover, the likelihood of the presence of `null` values when mapping XML data to relations motivated us to adopt a slightly different semantics for FD satisfaction, which defines an explicit treatment for `null` values.

6.3.2 The Correctness of the Propagation Algorithm

We next verify the correctness of the algorithm:

Theorem 6.3 *Given any set Σ of XML keys, a transformation σ and an FD ϕ over a relation R , Algorithm `propagation` returns `true` if and only if $\Sigma \models_{\sigma} R : \phi$.*

We will first give a rough outline of the proof. Let $\phi = Y \rightarrow Z$ be the FD to be checked. The proof can be divided in two parts: The first is to show that `Zcheck` is empty if and only if for any XML tree T such that $T \models \Sigma$, and for any tuples t_1 and t_2 in $\sigma(T)$ that do not contain `null` on attributes $Y \cup Z$, $\pi_Y(t_1) = \pi_Y(t_2)$ implies $\pi_Z(t_1) = \pi_Z(t_2)$. We will refer to this condition as the *agreement condition* for FD satisfaction. The second part is to show that `Ycheck` is empty if and only if and for any tuple t in $\sigma(T)$, $\pi_Y(t)$ contains `null` implies that $\pi_Z(t)$ also contains `null`. That is, the two conditions for FD satisfaction are guaranteed to hold in $\sigma(T)$ given that $T \models \Sigma$.

The proof of the first part is a bit long. The outline is as follows. First, we decompose $Y \rightarrow Z$ to a set of “simple” FDs $Y_i \rightarrow Z_i$, where each field in the set $Y_i \cup Z_i$ is defined along a path from x_r to a keyed variable x_i . Then, in Lemma 6.1 we will show that $\Sigma \models_{\sigma} Y \rightarrow Z$ if and only if for every i , $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$. Given this, we encode Y_i with a set of transitive keys Θ_i for the x_i -node, Z_i with a set of keys Γ_i determining the uniqueness of each field in Z_i , and reduce the problem of showing $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ to the problem of showing XML key implication $\Sigma \models \Theta_i \cup \Gamma_i$. That is, $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ if and only if $\Sigma \models \Theta_i \cup \Gamma_i$. The if direction requires Lemma 6.2, and the only if direction is established by Lemma 6.3. Therefore, given that algorithm **propagation** first looks for a transitive set of keys for a variable *target* which only involves attributes that define fields in Y , and when such a key is found, it removes from *Zcheck* fields that are unique for *target* or for an ancestor of *target*, we can conclude that the agreement condition of FDs is satisfied if and only if *Zcheck* becomes empty.

Example 6.11 Consider a transformation σ depicted as a table tree in Figure 6.6(b), and a set of XML keys Σ similar to those defined for Example 6.3.1:

- (k_1) $target^i$ is keyed with $@l_1^i$: $(Q_1^i/Q_2^i, \{ @l_1^i \})$;
- (k_2) z^i is unique for $target^i$: $(Q_1^i/Q_2^i, (Q_3^i/Q_4^i, \{ \}))$;
- (k_3) x_1^i is unique for x^i : $(Q_1^i, (Q_5^i, \{ \}))$;
- (k_4) $@l_2^i$ is required for y^i : $(Q_1^i/Q_2^i/Q_3^i, (\epsilon, S))$, and $@l_2^i \in S$. That is, there exists a key for the y^i -node involving attribute $@l_2^i$.

Consider the FD $\phi = l_{y_1}^1, l_{y_2}^1, \dots, l_{y_1}^k, l_{y_2}^k \rightarrow l_1^1, l_z^1, \dots, l_1^k, l_z^k$. The process of verifying whether ϕ is propagated from Σ can be reduced to checking the propagation of each $\phi_i = l_{y_1}^i, l_{y_2}^i \rightarrow l_1^i, l_z^i$, where ϕ_i is defined along a path from the root to $target^i$. Recall that $var(Z)$ denotes the set of variables that populate fields in Z . We have to show that *Zcheck* becomes empty if and only if every $z \in var(Z)$ is either unique for some $target^i$ (by checking keys of the form k_2), or unique for some ancestor of $target^i$ (by checking keys of the form k_3). Moreover, $target^i$ must be keyed by attributes that populate fields in Y , as exemplified by keys of the form k_1 .

Observe that keys of the form k_4 only define “extraneous” attributes on the left-hand side of ϕ and therefore are not relevant for the first part of the proof. However, they are relevant for the second part. Observe that if $@l_2^i$ were not an attribute of an ancestor of some $z \in \text{var}(Z)$ or were not required to exist, then there is no guarantee that whenever fields in Z are defined, $l_{y_2}^i$, the attribute populated by $@l_2^i$, is also defined. This is why algorithm **propagation** removes from $Ycheck$ all the required attributes of all $z \in \text{var}(Z)$ and of all their ancestors. \square

Before developing the details of the first part of the proof, we present a few definitions.

Definition 6.2 *Let $\text{Rule}(R)$ be a transformation rule, and Σ a set of XML keys. We say that a label l is required for a variable x (according to Σ) if $l : \text{value}(y)$ in $\text{Rule}(R)$, $y \leftarrow x'/@a$, $x' = x$ or x' is an ancestor of x , and there exists an XML key $(Q, (Q', S)) \in \Sigma$ such that $P(x_r, x') \subseteq Q/Q'$ and $@a \in S$.*

Definition 6.3 *Let $\text{Rule}(R)$ be a transformation rule, Y a set of fields in $\text{Attr}(R)$, and w a variable in $\text{Rule}(R)$. We denote by:*

- *$Yattr(w)$: the set of attributes of w that populate fields in Y . That is, $Yattr(w) = \{@a \mid y \leftarrow w/@a, l : \text{value}(y), \text{ and } l \in Y\}$.*
- *$\text{var}(Y)$: the set of variables corresponding to the set of labels Y . That is, $\text{var}(Y) = \{y \mid l : \text{value}(y), l \in Y\}$.*
- *$\text{varInPath}(Y)$: the set of variables along the path from x_r to some y in $\text{var}(Y)$. That is, $\text{varInPath}(Y) = \{y' \mid l : \text{value}(y), y \in \text{var}(Y), P(x_r, y')/P(y', y) = P(x_r, y), y' \neq y\}$.*

Proof of Part 1. We have to show that $Zcheck$ becomes empty if and only if for any XML tree T such that $T \models \Sigma$, $\sigma(T)$ satisfies the agreement condition for FD satisfaction.

First, observe that if $Z \subseteq Y$ then by algorithm **propagation**, $Zcheck$ becomes empty, and it is obvious that for any XML tree T , $\sigma(T) \models \phi$ with respect to the agreement condition. When $Z \not\subseteq Y$, we will first show that checking propagation of ϕ is equivalent

to checking propagation on FDs $\{\phi_1, \dots, \phi_n\}$, where each ϕ_i is a simple FD. Recall that $varInPath(Y)$ denotes the set of variables along a path from x_r to a variable in $var(Y)$. Let us denote by $pathsDefBy(Y)$ the longest paths defined by variables in $varInPath(Y)$. That is, $pathsDefBy(Y) = \{W_1, \dots, W_n\}$, where each W_i corresponds to a path from x_r to a variable in $varInPath(Y)$ with no descendants in the same set. More specifically, $W_i = \{y_1, \dots, y_k\}$, where for every $i \in [1, k]$, $y_i \leftarrow y_{i-1}/Q_i$ is a mapping in σ , $y_0 = x_r$, and there exists no $y' \in (varInPath(Y) \setminus W_i)$ defined with $y' \leftarrow y_k/Q'$, for some path Q' . Note that $\cup_{i \in [1, n]}(W_i) = varInPath(Y)$. Now we are in position to define ϕ_i : $\phi_i = Y_i \rightarrow Z_i$, where $Y_i \subseteq Y$ is the set of labels populated with attributes of variables in W_i , and $Z_i \subseteq Z$ is the set of labels populated with variables such that its the closest ancestor w with $Yattr(w) \neq \emptyset$ is in W_i . We denote this set of FDs as $fdsDefBy(\phi)$. The Lemma below shows that $\Sigma \models_{\sigma} \phi$ if and only if $\Sigma \models_{\sigma} fdsDefBy(\phi)$.

Lemma 6.1 *Let Σ be a set of XML keys, σ a transformation, and ϕ an FD. $\Sigma \models_{\sigma} \phi$ if and only if $\Sigma \models_{\sigma} fdsDefBy(\phi)$.*

Proof. Let $\phi = Y \rightarrow Z$, and $fdsDefBy(\phi) = \{Y_i \rightarrow Z_i \mid i \in [1, n]\}$. The if direction is direct. That is, given that for all $i \in [1, n]$ $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$, by the augmentation and transitivity rules for FD inference we can obtain $Y \rightarrow Z$. Therefore, $\Sigma \models_{\sigma} Y \rightarrow Z$. For the other direction, we will show that if $\Sigma \models Y \rightarrow Z$ then every ϕ_i is well-defined, that is, it is not of the form $Y_i \rightarrow \emptyset$, and moreover, for each Z_i , Y_i is the largest subset of Y such that $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$.

Recall that $fdsDefBy(\phi)$ is defined based on $pathsDefBy(Y)$, which denote the longest paths from the root to a parent of a variable in $var(Y)$. Let $pathsDefBy(Y) = \{W_1, \dots, W_n\}$, where the last variable in each path W_i is x_i . First, we will show that ϕ_i is well-defined. More specifically, we will show that in each ϕ_i there exists at least one variable $z_i \in Z_i$ that is a descendent of x_i . Suppose, by contradiction, that there exists a ϕ_j that does not satisfy this condition. Then we can build an XML tree T such that $T \models \Sigma$ but $T \not\models \phi$, that is $\Sigma \not\models_{\sigma} \phi$, which leads to a contradiction. The construction of T is as follows. T consists of a single node corresponding to every variable in $var(Y) \cup varInPath(Y) \cup var(Z)$, except for x_j and $Yattr(x_j)$. Clearly, $T \models \Sigma$. But $\sigma(T)$ consists of a single tuple t such that $\pi_Y(t)$

contains `null` but not $\pi_Z(t)$. This is because, by definition, $Yattr(x_j) \neq \emptyset$. Since in T a node corresponding to x_j is missing, so is $Yattr(x_j)$. Therefore, all fields in Y populated with $Yattr(x_j)$ contain `null` and $\sigma(T) \not\models \phi$, which contradicts our assumption.

Given that every ϕ_i is well-defined, we are now going to show that in each ϕ_i , there exists no set larger than Y_i such that $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$. Consider an arbitrary Z_i , and suppose by contradiction that there exists a set $S \subset Y$, such that $\Sigma \models_{\sigma} Y_i \cup S \rightarrow Z_i$, where $S \neq \emptyset$ and $S \cap Y_i = \emptyset$. Consider an XML tree T constructed as follows. T consists of a single node corresponding to each variable in W_i and $var(Z_i)$, and each node contains all its required attributes. All attributes and elements in T are given the same value. Obviously, $T \models \Sigma$. Also, $\sigma(T)$ contains a single tuple t , where $\pi_{Z_i}(t)$ does not contain `null`. But there are no nodes in T to populate fields in S , since by construction fields in S are not required for variables in W_i . Therefore, $\pi_{Y_i \cup S}(t)$ contains `null`. Therefore, $\Sigma \not\models_{\sigma} R : Y_i \cup S \rightarrow Z_i$, a contradiction. This shows that fields in Z_i do not depend on any field in $Y \setminus Y_i$. Thus, if $\Sigma \models_{\sigma} \phi$, it must be the case that for every $i \in [1, n]$, $\Sigma \models_{\sigma} \phi_i$, and ϕ was obtained from these FDs by the augmentation and transitivity rules for FD inference. \square

Given that checking whether ϕ is propagated from Σ is equivalent to checking if every ϕ_i is propagated from Σ , we will now define sets of keys Θ_i and Γ_i such that $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ if and only if $\Sigma \models \Theta_i \cup \Gamma_i$. Let Θ_i be the transitive set of keys computed for x_i by algorithm `propagation`. That is, $\Theta_i = \{\kappa_1, \dots, \kappa_m\}$, where $\kappa_j = (P(x_r, y_{j-1}), (P(y_{j-1}, y_j), att[y_j]))$, $y_0 = x_r$, and $y_m = x_i$. Let Γ_i denote the set of keys determining the uniqueness of fields in Z_i . That is, $\Gamma_i = \{(P(x_r, x'), (P(x', z), \{\})) \mid x' = x_i \text{ or } x' \text{ is an ancestor of } x_i, \text{ and } z \in var(Z_i)\}$.

(a) $\Sigma \models \Theta_i \cup \Gamma_i$ implies $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ with respect to the agreement condition

Recall that Θ_i is a transitive set of keys for an x_i -node, where each key is defined only with attributes $@A_j$ that define fields in Y_i . We will show that if $\Sigma \models \Theta_i$ then for any XML tree T that satisfies Σ there exists no two nodes in $\llbracket P(x_r, x_i) \rrbracket$ that agree on the values of all attributes $@A_j$. Since every field in Z_i is defined with unique nodes along this path, we will then show that in $\sigma(T)$ there exists no two two tuples that agree on Y_i but not in Z_i .

First, we will formalize the notion of two nodes in an XML tree T *agreeing on* a set of labels Y . Let $\text{Rule}(R)$ be a transformation, Y a set of fields in $\text{Attr}(R)$, and $\{y_1, \dots, y_m\}$ be variables defined in $\text{Rule}(R)$ along a path $P(x_r, y_m)$ with $Y_{\text{attr}}(y_i) \neq \emptyset$. Let T be an XML tree, and $v_0, \dots, v_m, v'_0, \dots, v'_m$ be two sets of nodes in T such that $v_0, v'_0 = r$, and for all $i \in [1, m]$, $v_i \in v_{i-1} \llbracket P(y_{i-1}, y_i) \rrbracket$, and $v'_i \in v'_{i-1} \llbracket P(y_{i-1}, y_i) \rrbracket$. We say that v_m and v'_m *agree on* Y if for every i in $[1, m]$ and every $@a$ in $Y_{\text{attr}}(y_i)$, $\text{val}(v_i.@a) = \text{val}(v'_i.@a)$.

Having presented the definition, the next Lemma establishes that it is indeed the case that $\Sigma \models \Theta_i$ implies that for any XML tree T that satisfies Σ there exists no two nodes agreeing on Y_i .

Lemma 6.2 *Let $\text{Rule}(R)$ be a transformation, Y a set of fields in $\text{Attr}(R)$, and $\{y_1, \dots, y_m\}$ be variables defined in $\text{Rule}(R)$ along a path $P(x_r, y_m)$ with $Y_{\text{attr}}(y_i) \neq \emptyset$. Let Θ be a set of transitive keys $\{\kappa_1, \dots, \kappa_m\}$, where $\kappa_i = (P(x_r, y_{i-1}), (P(y_{i-1}, y_i), Y_{\text{attr}}(y_i)))$, $i \in [1, m]$, and $y_0 = x_r$. Let Y_i be the set of fields in Y defined by attributes in $Y_{\text{attr}}(y_j)$, $j \leq i$. Then $\Sigma \models \Theta$ if and only if for any XML tree T such that $T \models \Sigma$ there exist no two nodes in $\llbracket P(x_r, y_i) \rrbracket$ agreeing on Y_i , for all $i \in [1, m]$.*

Proof. First, suppose, by contradiction, that for any XML tree T that satisfies Σ there exist no two distinct nodes agreeing on any Y_i , yet $\Sigma \not\models \Theta$. That is, there exists a key $\kappa_i \in \Theta$ such that $\Sigma \not\models \kappa_i$. Therefore, there must exist an XML tree T' such that $T' \models (\Sigma \wedge \neg \kappa_i)$. But if $T' \not\models \kappa_i$, it must contain two distinct nodes n, n' in $\llbracket P(x_r, y_i) \rrbracket$ agreeing on Y_i , a contradiction. Thus, $\Sigma \models \Theta$.

For the other direction, let T be a tree that satisfies Σ , and suppose $\Sigma \models \Theta$. We will show by induction on m , the size of Θ , that there exist no two distinct nodes $\llbracket P(x_r, y_m) \rrbracket$ in T agreeing on Y_m . For the base case, $m = 1$, $\Theta = \{(\epsilon, (P(x_r, y_1), Y_{\text{attr}}(y_1)))\}$. That is, it consists of a single absolute key. Since $T \models \Sigma$, and $\Sigma \models \Theta$, T cannot have two distinct nodes in $\llbracket P(x_r, y_1) \rrbracket$ agreeing on Y_1 . Assume the statement for $m - 1$, and we will show that it also holds for m . Let w be an arbitrary node in $\llbracket P(x_r, y_{m-1}) \rrbracket$. By inductive hypothesis, there exists no other node that agrees on Y_{m-1} with w . Therefore, it suffices to show that there exist no two nodes u, u' in $w \llbracket P(y_{m-1}, y_m) \rrbracket$ agreeing on $Y_{\text{attr}}(y_m)$. By

definition, Θ contains a key $(P(x_r, y_{m-1}), (P(y_{m-1}, y_m), Yattr(y_m)))$, and by assumption, $\Sigma \models \Theta$; hence $T \models \Theta$. Since u and u' agree on $Yattr(y_m)$, they cannot be distinct nodes in T . This completes the proof. \square

Given that in any XML tree T that satisfies Σ there exists no two nodes in $\llbracket P(x_r, x_i) \rrbracket$ agreeing on Y_i whenever $\Sigma \models \Theta_i$, we will next show that if $\Sigma \models \Theta_i \cup \Gamma_i$ then in $\sigma(T)$ there are no two tuples t_1, t_2 that do not contain `null` on fields in Y_i such that $\pi_{Y_i}(t_1) = \pi_{Y_i}(t_2)$ and $\pi_{Z_i}(t_1) \neq \pi_{Z_i}(t_2)$. Suppose, by contradiction, that there exists two such tuples t_1 and t_2 . Let $ancestors(x_i) = \{y_1, \dots, y_k\}$. By the semantics of transformation σ , if a tuple t is in $\sigma(T)$ then there are nodes v_1, \dots, v_k on a path $P(x_r, x_i)$ in T such that y_i is mapped to v_i , v_1 is the root r , $v_k \in \llbracket P(x_r, x_i) \rrbracket$, and $v_i \in v_{i-1} \llbracket P(y_{i-1}, y_i) \rrbracket$ for $i \in [2, k]$. Moreover, for each $l \in Y_i$, $t.l$ is determined by $val(v_i.@a)$ for some $@a \in Yattr(y_i)$, and for each $l \in Z_i$, $t.l$ is determined by the value of a node in $v_i \llbracket Q \rrbracket$. Since $\Sigma \models \Gamma_i$, for each v_i , $|v_i \llbracket Q \rrbracket| \leq 1$. Thus one can easily verify that the existence of two tuples t_1 and t_2 such that $\pi_{Y_i}(t_1) = \pi_{Y_i}(t_2)$ and $\pi_{Z_i}(t_1) \neq \pi_{Z_i}(t_2)$ implies the existence of two distinct nodes $u, u' \in \llbracket P(x_r, x_i) \rrbracket$ agreeing on Y , which contradicts Lemma 6.2. Thus, $\Sigma \models (\Theta_i \cup \Gamma_i)$ implies $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ with respect to the agreement condition.

(b) $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ with respect to the agreement condition implies $\Sigma \models \Theta_i \cup \Gamma_i$

First, observe that Y_i is defined as consisted of all fields in Y that are populated with attributes of nodes in a path $P(x_r, x_i)$, no matter if they are necessary for identifying a node in $\llbracket P(x_r, x_i) \rrbracket$ or not. Thus, Y_i may contain extraneous fields. If this is the case, there exists a subset Y'_i of Y_i such that $\Sigma \models_{\sigma} R : Y'_i \rightarrow Z_i$. We denote as Y_{min} a minimum subset of Y_i such that $\phi' = Y_{min} \rightarrow Z_i$, and $\Sigma \models_{\sigma} \phi'$.

If $\Sigma \models_{\sigma} \phi'$ then there exists a transitive set of keys Θ'_i such that $\Sigma \models \Theta'_i$ and a set of keys determining the uniqueness of each label in Z_i , that coincides with Γ_i , such that $\Sigma \models \Gamma_i$. This is asserted by Lemma 6.3, given below. Therefore, to prove the claim, it suffices to show that if $\Sigma \models \Theta'_i$ then $\Sigma \models \Theta_i$. This is a direct consequence of Lemma 6.2. More specifically, let T be an arbitrary XML tree such that $T \models \Sigma$. Since, $\Sigma \models \Theta'_i$ then by Lemma 6.2, there exist no two nodes in $\llbracket P(x_r, x_i) \rrbracket$ agreeing on Y_{min} . Since $Y_{min} \subseteq Y_i$, it

cannot be the case that two nodes agree on Y_i but not on Y_{min} . Thus, by Lemma 6.2 and the construction of Θ_i , $\Sigma \models \Theta_i$. Recall that in algorithm `propagation` a label l is removed from $Zcheck$ whenever a transitive set of keys for a variable $target$ is found, and l is either unique for $target$ or for some ancestor of $target$. Thus, $Zcheck$ becomes empty if and only if $Y \rightarrow Z$ satisfies the agreement condition of FDs.

Lemma 6.3 *Let Σ be a set of XML keys, σ be a transformation with a corresponding table tree T_R , and $V = \{y_1, \dots, y_k\}$ be a set of variables such that $y_i \leftarrow y_{i-1}/P(y_{i-1}, y_i)$ is a mapping in σ , and $y_0 = x_r$.*

Let $\phi = Y \rightarrow Z$ be an FD such that every $l \in Y$ is a required label for some $y_i \in V$, and $V' = \{y'_1, \dots, y'_m\}$ be the set consisted of all variables in V with $Yattr(y_i) \neq \emptyset$. If $\Sigma \models_\sigma \phi$, $Z \not\subseteq Y$, and for no set $Y' \subset Y$, $\Sigma \models_\sigma Y' \rightarrow Z$ then:

1. *there exists a transitive set of keys $\Theta = \{\kappa_1, \dots, \kappa_m\}$, where κ_i is of the form $(P(x_r, y'_{i-1}), (P(y'_{i-1}, y'_i), Yattr(y'_i)))$, $y'_0 = x_r$, and $\Sigma \models \Theta$.*
2. *there exists $l \in Z$ defined by $l : value(z)$ such that $\Sigma \models (P(x_r, y'_m), (P(y'_m, z), \{\}))$; moreover, for every $l' \in (Z - \{l\})$, if l' is defined by $l' : value(z')$, and y_i is the closest ancestor of z' in V then $\Sigma \models (P(x_r, y_i), (P(y_i, z'), \{\}))$.*

Proof. Let $\Gamma = \{(P(x_r, w), (P(w, z), \{\})) \mid \text{where } z \in var(Z), \text{ and } w \text{ is the closest ancestor of the } z \text{ in } V\}$. It suffices to show that if $\Sigma \not\models \Theta \cup \Gamma$ then $\Sigma \not\models_\sigma Y \rightarrow Z$. More specifically, given that $\Sigma \not\models \Theta \cup \Gamma$, there exists an XML tree T such that $T \models \Sigma$ but $T \not\models \Theta \cup \Gamma$. Then, we can show that in $\sigma(T)$ there exists two tuples t_1 , and t_2 such that $\pi_Y(t_1) = \pi_Y(t_2)$ but $\pi_Z(t_1) \neq \pi_Z(t_2)$. That is, $\Sigma \not\models_\sigma Y \rightarrow Z$.

First, suppose that $\Sigma \models \Gamma$, but $\Sigma \not\models \Theta$. Let y'_{n+1} be the first variable in V' such that $\Sigma \not\models \kappa_{n+1}$. That is, for all $j \leq n$, $\Sigma \models \kappa_j$, yet $\Sigma \not\models \kappa_{n+1}$. Next, we describe the construction of an XML tree T such that $T \models (\Sigma \wedge \Gamma \wedge \neg\Theta)$. T contains a single path Q from the root to a node u in $\llbracket P(x_r, y'_n) \rrbracket$, and two paths from u to nodes u', u'' in $u \llbracket P(y'_n, y'_m) \rrbracket$. Every node in T has all the required attributes according to Σ . Let $var(Z) = \{z_1, \dots, z_l\}$, and $\{w_1, \dots, w_l\}$ be the closest ancestors in V of each z_i . If w_i is along the path $P(x_r, y'_n)$

then a single node attached to the unique node in $\llbracket P(x_r, w_i) \rrbracket$ is created. Otherwise, a node corresponding to each z_i is created in each subtree rooted at u . Arbitrary values are assigned to attributes in the path Q . Corresponding attributes in both subtrees rooted at u have the same value if they populate fields in Y and different values otherwise. In other words, if u_j, u'_j are in $n\llbracket P(y'_n, y'_j) \rrbracket$, then $val(u_j.\@a) = val(u'_j.\@a)$ if $\@a$ in $Yattr(y'_j)$, and $val(u_j.\@a) \neq val(u'_j.\@a)$, otherwise.

We will now show that T indeed satisfies Σ and Γ , but not Θ . It is easy to see that $T \models \Gamma$ since at most one node corresponding to each $z \in var(Z)$ is created at a single node. It is also easy to see that $T \not\models \Theta$. Before showing that $T \models \Sigma$, recall that by assumption the set Y in ϕ is minimum. That is, for no set $Y' \subset Y$, $\Sigma \models_{\sigma} Y' \rightarrow Z$. Since every label in Y corresponds to a key path in Θ , Θ is a minimum transitive set of keys for y'_m . Based on this observation, we will show that $T \models \Sigma$. Since in T there exists a single path Q from the root to u , T can only violate Σ if it is one of the following cases:

(1) the set of attributes $Yattr(y'_j)$, $j > n + 1$, identifies a node in the subtree rooted at u , that is, $\Sigma \models (Q, (P(y'_n, y'_j), Yattr(y'_j)))$: Observe that there exists a key κ_j in Θ , where $\kappa_j = (Q/P(y'_n, y'_{j-1}), (P(y'_{j-1}, y'_j), Yattr(y'_j)))$. Since Θ is minimum, it cannot be the case that $\Sigma \models (Q, (P(y'_n, y'_j), Yattr(y'_j)))$, otherwise the set of attributes $Yattr(y'_{j-1})$, relative to $P(x_r, y'_{j-1})$ would be extraneous. Therefore, $\Sigma \not\models (Q, (P(y'_n, y'_j), Yattr(y'_j)))$, for any $j > n + 1$.

(2) a node in a subtree under u must be unique for u , that is, $\Sigma \models (Q, (P(y'_n, w), \{\}))$ for some $w \in (V \cup var(Y) \cup var(Z))$: Suppose $P(y'_n, w) = P_1/P_2$, where Q/P_1 is the path in the table tree from x_r to the closest ancestor of w that is in V . To see that $\Sigma \not\models (Q, (P_1/P_2, \{\}))$, observe that if $\Sigma \models (Q, (P_1/P_2, \{\}))$, then by the *target-to-context* rule for XML key inference, $\Sigma \models (Q/P_1, (P_2, \{\}))$. Therefore, the attributes $Yattr(y'_n)$ relative to y'_n would be extraneous, and Θ would not be minimum.

Given that $T \models \Sigma \wedge \Gamma \wedge \neg\Theta$, we next verify that $\sigma(T)$ consists of two tuples that violates the agreement condition for FD satisfaction. By the semantics of transformation σ , the two tuples t_1, t_2 in $\sigma(T)$ are such that: (a) for each $l' \in Y$, $t_1.l'$ and $t_2.l'$ are the values of the same node reached by following $P(x_r, y'_i/\@a)$, for $1 \leq i \leq n$, or corresponding nodes in the subtrees rooted at u . In both cases, $t_1.l' = t_2.l'$; (b) for the distinguished label $l \in Z$

populated with a descendent of y'_m , $t_1.l$ and $t_2.l$ are the values of some descendent of u' and u'' such that $t_1.l \neq t_2.l$, since by construction $l \notin Y$. Thus t_1 and t_2 are indeed two tuples in $\sigma(T)$ that violate FD ϕ .

We now turn to the case that $\Sigma \models \Theta$, but $\Sigma \not\models \Gamma$. Similar to the previous discussion, we will construct an XML tree T such that $T \models \Sigma \wedge \Theta \wedge \neg\Gamma$, and show that $\sigma(T) \not\models Y \rightarrow Z$. Let z be an arbitrary variable in $\text{var}(Z)$ such that $\Sigma \not\models \varphi$, where $\varphi = (P(x_r, w), (P(w, z), \{\}))$, and w is the closest ancestor of z in V . Consider an XML tree T constructed as follows. T consisted of a single path from the root to a node u reached by following path $P(x_r, y_k)$. Each node along the path contains all the required attributes, and, additionally, one node corresponding to each variable in $\text{var}(Z) \setminus \{z\}$. The node corresponding to $P(x_r, w)$ contains two descendents u, u' reached by following path $P(w, z)$, and distinct values are assigned to each of them; that is, $\text{val}(u) \neq \text{val}(u')$. Arbitrary values are assigned to every other node. It is easy to see that $T \models (\Sigma \wedge \Theta \wedge \neg\varphi)$. We will now show that $\sigma(T)$ does not satisfy ϕ . Let $l_z \in Z$ be the field populated with z . By the semantics of transformations, $\sigma(T)$ contains two tuples t_1, t_2 , such that $t_1.l_z$ corresponds to the value the u node, $t_2.l_z$ corresponds to u' , and all other fields in t_1 and t_2 correspond to the value of the same node in T . Therefore, $\pi_{Y \cup Z \setminus \{l_z\}}(t_1) = \pi_{Y \cup Z \setminus \{l_z\}}(t_2)$, but $\pi_{l_z}(t_1) \neq \pi_{l_z}(t_2)$. That is, $\Sigma \not\models_{\sigma} Y \rightarrow Z$, as we wanted to prove. \square

Proof of Part 2. Next, we have to show that $Ycheck$ is empty if and only if for any XML tree T such that $T \models \Sigma$, and for any tuple t in $\sigma(T)$, $\pi_Y(t)$ contains **null** implies that $\pi_Z(t)$ also contains **null**.

For the if direction, assume that $Ycheck$ is empty, T is an XML tree that satisfies Σ , and suppose, by contradiction, that in $\sigma(T)$ there exists a tuple t such that $\pi_Y(t)$ contains **null** but not $\pi_Z(t)$. Let l_a be an arbitrary field in Y such that $\pi_{l_a}(t)$ is **null**, where l_a is specified with $l_a : \text{value}(x_a)$ and $x_a \leftarrow x_b/@a$. Observe that x_b must be an ancestor of some $z \in \text{var}(Z)$, otherwise, by algorithm **propagation**, $Ycheck \neq \{\}$. Let $l_z : \text{value}(z)$. By the semantics of transformations, if $\pi_{l_a}(t) = \mathbf{null}$, and $\pi_{l_z}(t) \neq \mathbf{null}$ then in T there exists a node $v \in \llbracket P(x_r, x_b) \rrbracket$, and $n \in v \llbracket P(x_b, z) \rrbracket$ such that v does not have an attribute $@a$. But since $@a$ is an attribute of an ancestor of x , and $Ycheck$ is empty, $@a$ must be

a required attribute for x_b . Therefore, $T \not\models \Sigma$, which contradicts our assumption. We conclude that $Ycheck = \{\}$ implies that for any tuple t in $\sigma(T)$, if $\pi_Y(t)$ contains **null** then so does $\pi_Z(t)$. For the other direction, suppose $\Sigma \models_{\sigma} Y \rightarrow Z$. We have to show that $Ycheck$ is empty. Recall that in algorithm `propagation` $Ycheck$ is initialized with $(Y \setminus required[x_r])$ and it is subsequently subtracted by every required attribute of variables in $var(Z)$ and their ancestors. Therefore, it suffices to show that indeed whenever $\Sigma \models_{\sigma} \phi$ every field in Y is required for one of these variables. This is established by the following Lemma.

Lemma 6.4 *Let Σ be a set of XML keys and σ a transformation rule. If $\Sigma \models_{\sigma} Y \rightarrow Z$ then for every $l \in Y$, l is required either for some $z \in var(Z)$ or for an ancestor of some $z \in var(Z)$.*

Proof. Recall that $varInPath(Z)$ denotes the set of variables along a path from x_r to a variable in $var(Z)$. The proof is by contradiction. That is, assume that there exists a label l in Y such that l is not required for any z in $var(Z) \cup varInPath(Z)$. Then we will show that there exists an XML tree T such that $T \models \Sigma$ and $\sigma(T) \not\models Y \rightarrow Z$. Let y be the variable that populates field l . Consider an XML tree T constructed as follows. T consists of a single node corresponding to each variable in $varInPath(Z)$ and $var(Z)$, and each node contains all its required attributes according to Σ . All attributes and elements in T are given the same value. Obviously, $T \models \Sigma$. Also, $\sigma(T)$ contains a single tuple t , where $\pi_Z(t)$ does not contain **null**. But since $[[P(x_r, y)]] = \{\}$, $\pi_l(t) = \mathbf{null}$. Therefore, $\Sigma \not\models_{\sigma} Y \rightarrow Z$, a contradiction. \square

Thus, if $\Sigma \models_{\sigma} Y \rightarrow Z$, every field in Y is required either for a variable that defines a field in Z or for one of its ancestors. Thus, by the computation of $Ycheck$, the variable becomes empty at the end of the algorithm `propagation`. This completes the proof. \square

6.4 Computing Minimum Cover

In this section we present two algorithms for finding a minimum cover for FDs propagated from XML keys. The first algorithm is a direct generalization of Algorithm `propagation` of Figure 6.7, and always takes exponential time. We use this naive algorithm to illustrate the difficulties in connection with finding a minimum cover. The second algorithm takes polynomial time in the size of input, by reducing the number of FDs generated in the following way: a new FD is inserted into the resulting set only if it cannot be implied from the FDs already generated, using the inference rules for FDs. To the best of our knowledge, this is the first effective algorithm for finding a minimum cover for FDs propagated from XML keys.

We present the two algorithms in Sections 6.4.1 and 6.4.2 respectively; a correctness proof is provided in Section 6.4.3.

6.4.1 A Naive Algorithm

Algorithm `propagation` given in the last section allows us to check XML key propagation. Thus a naive algorithm for finding a minimum cover is to generate each possible FD on \mathbf{U} , check whether or not it is in F^+ , the set of all the FDs mapped from the XML keys, using Algorithm `propagation`, and then eliminate redundant FDs from F^+ using a minimization algorithm; this yields a minimum cover F_{mc} for F^+ . The algorithm, Algorithm `naive`, is shown in Figure 6.8. It takes exponential time in the size of \mathbf{U} for any input since it computes all possible FDs on \mathbf{U} .

Obviously, Algorithm `naive` is too expensive to be practical. The problem with the algorithm is that it needs to compute F^+ , which is exponential in the size of \mathbf{U} .

Example 6.12 Consider the transformation depicted as the table tree in Figure 6.9(a), and a set Σ consisting of $2k + 1$ keys: $(a_1/\dots/a_{i-1}, (a_i, \{\@l_i^j\}))$ for $i \in [1, k]$ and $j \in [1, 2]$ plus $(a_1/\dots/a_k, (l, \{\}\))$. Then F^+ includes the following: (1) 2^k many FDs of the form $l_1^{j_1}, \dots, l_k^{j_k} \rightarrow l$, where $j_i \in [1, 2]$ for $i \in [1, k]$; (2) for each $m \in [1, k]$, 2^m many FDs of

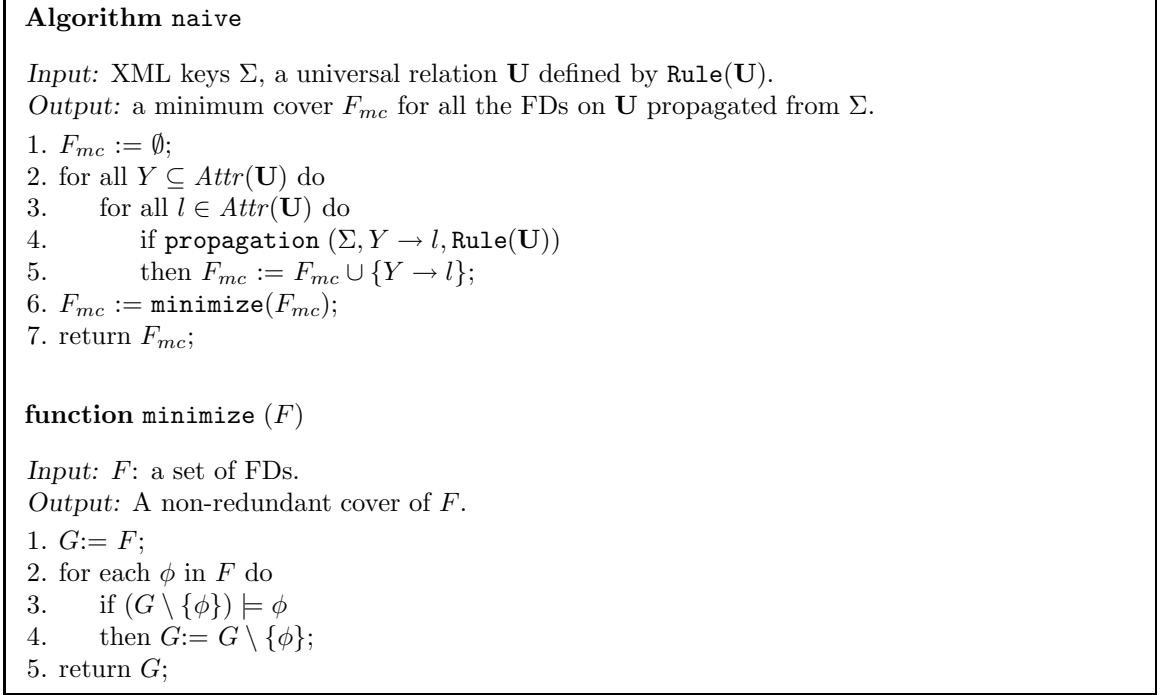


Figure 6.8: A naive algorithm for finding a minimum cover

the form: $l_1^{j_1}, \dots, l_m^{j_m} \rightarrow l_m^{j'_m}$, where $j_m, j'_m \in [1, 2]$ and $j_m \neq j'_m$; (3) for each $m \in [1, k]$, $4(m-1)$ many FDs of the form $l_m^1, l_s^{j_s} \rightarrow l_m^1$ and $l_m^2, l_s^{j_s} \rightarrow l_m^2$, where $s < m$ and $j_s \in [1, 2]$. The set is exponentially large, although it does not include any trivial FDs.

A minimum cover F_{mc} for F^+ , however, consists of $2k+1$ FDs only (for $m \in [1, k]$):

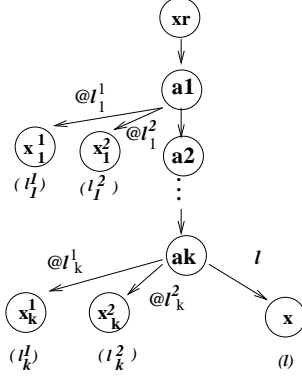
$$l_1^1, \dots, l_m^1 \rightarrow l_m^2, \quad l_1^1, \dots, l_{m-1}^1, l_m^2 \rightarrow l_m^1, \quad l_1^1, \dots, l_k^1 \rightarrow l.$$

Plus at most $4k^2$ FDs of the form $l_m^1, l_s^{j_s} \rightarrow l_m^1$ and $l_m^2, l_s^{j_s} \rightarrow l_m^2$. □

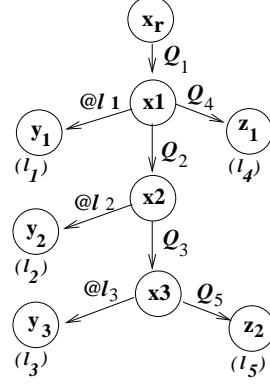
This observation motivates us to develop an algorithm that directly finds F_{mc} without computing F^+ .

6.4.2 A Polynomial-Time Algorithm

We next present a more efficient algorithm for finding a minimum cover for all the propagated FDs. The algorithm takes as input a set Σ of XML keys and a universal relation



(a) Table tree of Example 6.4.1



(b) Table tree of Example 6.4.2

Figure 6.9: Trees representing universal relations

\mathbf{U} defined by a transformation σ ; it computes a minimum cover for all FDs on \mathbf{U} propagated from Σ via σ in $O(m^4n^4)$ time, where m and n are the sizes of XML keys Σ , and transformation σ , respectively.

The algorithm works as follows. Recall that the transformation $\text{Rule}(\mathbf{U})$ can be depicted as a table tree $T_{\mathbf{U}}$, in which each variable x in the set X of $\text{Rule}(\mathbf{U})$ is represented by a unique node, referred to as the x -node. The algorithm traverses $T_{\mathbf{U}}$ top-down starting from the root x_r , and generates a set F_{mc} of FDs that is a minimum cover of F^+ . More specifically, at each x -node encountered, it expands F_{mc} by including certain FDs propagated from Σ .

The obvious question is what new FDs are added at each x -node. As in Algorithm **propagation**, at each x -node a new FD $Y \rightarrow Z$ is included in F_{mc} only if (1) x is keyed with a set of attributes that define the fields in Y ; (2) each field $l \in Z$ is defined by the value of a node y and y is unique for x ; (3) $Y \rightarrow Z$ cannot be derived from FDs previously inserted in F_{mc} using the rules for FD inference. The first condition requires that x has a keyed ancestor $target$ and a set S of attributes such that $\Sigma \models (P(x_r, target), (P(target, x), S))$, and moreover, the key for $target$ and S define the fields of Y . The second condition can be stated as $\Sigma \models (P(x_r, x), (P(x, y), \{\}))$. The third condition prevents redundant FDs to be inserted in F_{mc} .

Example 6.13 Recall the universal relation \mathbf{U} defined by the transformation σ and the set Σ of XML keys of Example 6.2.2, and the table tree depicted in Figure 6.5. An FD derived at the z_2 node is $\text{bookIsbn}, \text{chapNum}, \text{secNum} \rightarrow \text{secName}$. The left-hand side of the FD corresponds to a transitive set of keys for the z_s -node consisted of a section $@number$ which is an attribute of z_s , as well as a chapter $@number$ and a book $@isbn$, which are a key of z_s 's ancestor y_c . The right-hand side of the FD is defined by a node z_2 unique under z_s , by KS_5 in Σ . Thus the key for the z_s node actually consists of the key of its ancestor y_c as well as a key for *section* ($@number$) relative to y_c . \square

Example 6.14 Consider the transformation rule below, depicted as a table tree in Figure 6.9(b).

$$\begin{aligned} \text{Rule}(R) = \{ & l_1: \text{value}(y_1), \quad l_2: \text{value}(y_2), \quad l_3: \text{value}(y_3), \quad l_4: \text{value}(z_1), \quad l_5: \text{value}(z_2)\}, \\ & x_1 \leftarrow x_r/Q_1, \quad x_2 \leftarrow x_1/Q_2, \quad x_3 \leftarrow x_2/Q_3, \quad y_1 \leftarrow x_1/@l_1, \\ & y_2 \leftarrow x_2/@l_2, \quad y_3 \leftarrow x_3/@l_3, \quad z_1 \leftarrow x_1/Q_4, \quad z_2 \leftarrow x_3/Q_5 \end{aligned}$$

From the following set Σ of XML keys:

$$\begin{array}{lll} \phi_1 : (Q_1, \{@l_1\}) & \phi_2 : (Q_1, (Q_2/Q_3, \{@l_3\})) & \phi_3 : (Q_1/Q_2, \{@l_2\}) \\ \phi_4 : (Q_1/Q_2, (Q_3, \{\})) & \phi_5 : (Q_1, (Q_4, \{\})) & \phi_6 : (Q_1/Q_2/Q_3, (Q_5, \{\})) \end{array}$$

we can conclude that there exists a single transitive set of keys for the x_1 -node, consisted of key ϕ_1 , where attribute $@l_1$ defines field l_1 . Therefore, given that by key ϕ_5 , l_4 is unique for x_1 , at this node FD $l_1 \rightarrow l_4$ is included in F_{mc} . Similarly, there exists one transitive set of keys for the x_2 -node consisted of ϕ_3 , which defines field l_2 . From keys ϕ_4 and ϕ_5 we can derive $(Q_1/Q_2, (Q_3/Q_5, \{\}))$ by the *context-to-target* rule for XML key inference. That is, l_5 is unique for x_2 . Moreover, by the *attribute* rule, it is true that $(Q_1/Q_2/Q_3, (@l_3, \{\}))$, and therefore, by the *context-to-target* rule, $\Sigma \models (Q_1/Q_2, (Q_3/@l_3, \{\}))$. That is, l_3 is also unique for x_2 . Thus, at this node, FDs $l_2 \rightarrow l_3$ and $l_2 \rightarrow l_5$ are included in F_{mc} . For the x_3 -node, there are two transitive sets of keys: $\{\phi_1, \phi_2\}$ and $\{\phi_3, \phi_4\}$, which define sets of fields $\{l_1, l_3\}$, and $\{l_2\}$, respectively. Since both l_3 and l_5 are unique for x_3 and FDs with l_2 on the left-hand side of the FD have already been included in F_{mc} , at this node only

one FD $\{l_1, l_3\} \rightarrow l_5$ is inserted in F_{mc} . \square

Critical to the performance of the algorithm is to minimize the number of FDs added at each x -node while ensuring that no FDs in F_{mc} is missed. This is done in two ways: First, we reduce our search for candidate FDs to those whose left-hand side corresponds to attributes of keys in Σ . Second, we observe that an ancestor *target* of an x -node may have several keys, but that in creating a transitive key for x only one of them must be selected as long as the following property is enforced: for any two transitive keys K_1 and K_2 of the x -node, F includes $Y_1 \rightarrow Y_2$, and $Y_2 \rightarrow Y_1$, where Y_1, Y_2 are sets of \mathbf{U} fields defined by K_1 and K_2 , respectively. Following Example 6.4.2, suppose an extra field l'_1 is defined in \mathbf{U} with $l'_1 : value(y'_1)$, where $y'_1 \leftarrow x_1/@l'_1$, and a new key $\phi'_1 = (Q_1, \{@l'_1\})$ is inserted in Σ . In this case, x_1 would have two transitive sets of keys consisted of keys $\{\phi_1\}$ and $\{\phi'_1\}$ and the following FDs would be generated at x_1 : $l_1 \rightarrow l_4$, $l_1 \rightarrow l'_1$, $l'_1 \rightarrow l_4$, and $l'_1 \rightarrow l_1$. Therefore, when creating a key for x_3 , based on ϕ_2 , we can select either l_1 or l'_1 to combine with l_3 . Recall that the only FD generated at x_3 is $\{l_1, l_3\} \rightarrow l_5$. Since $l'_1 \rightarrow l_1$, from these two FDs we can derive $\{l'_1, l_3\} \rightarrow l_5$.

There are some subtleties caused by the troublesome `null` value. First, if Y_1, Y_2 are sets of \mathbf{U} fields defined by two transitive keys K_1 and K_2 for x , then $Y_1 \rightarrow Y_2$, and $Y_2 \rightarrow Y_1$ are sound only if both contain at least one field defined by an attribute of the x -node. Consider again Example 6.4.2 and the sets of fields $\{l_2\}$ and $\{l_1, l_3\}$ that define two transitive sets of keys for the x_3 -node. Although it is indeed the case that $\Sigma \models_{\sigma} l_2 \rightarrow \{l_1, l_3\}$, it is *not* true that $\Sigma \models_{\sigma} \{l_1, l_3\} \rightarrow l_2$. This is because it is possible that in an XML tree a node corresponding to x_2 exists, without a node that corresponds to x_3 . If this is the case, the l_3 field contains `null` but not l_2 , and therefore $\sigma(T)$ violates the null restriction for FD satisfaction.

Second, let K_1 be a transitive key for an x -node, Y_1 be the sets of \mathbf{U} fields defined by K_1 , and l be a field defined by the value of a node y such that y is unique for an ancestor x' of x . Since there is a unique ancestor x' in a tree that connects to x via the path $P(x', x)$, a key for x is also a key for x' , provided that x exists under x' . Therefore, given that y is unique for x' , $Y_1 \rightarrow l$. Consider again Example 6.4.2, where l_2 is the field defined by key

ϕ_3 of the x_2 -node. Since in any XML tree, for each node $n_2 \in \llbracket P(x_r, x_2) \rrbracket$ there exists a single node n_1 such that $n_2 \in n_1 \llbracket P(x_1, x_2) \rrbracket$, and moreover, z_1 is unique for x_1 , we could expect that $\Sigma \models_{\sigma} l_2 \rightarrow l_4$. But since the existence of node n_1 does not guarantee the existence of a corresponding node n_2 , it is possible that in a tuple t of $\sigma(T)$, $t.l_2 = \mathbf{null}$ but $t.l_4 \neq \mathbf{null}$ therefore violating the null restriction for FD satisfaction. To overcome this problem, instead of $l_2 \rightarrow l_4$, an FD $l_2 \rightarrow \{l_2, l_4\}$ is inserted in F_{mc} .

Third, observe that an FD of the form $\{l_1, l_2\} \rightarrow l_1$ is sound if whenever the value of l_1 is not \mathbf{null} then the value of l_2 is also not \mathbf{null} . That is, if the value of l_1 is defined by the value of a node y , where y is a child of x , then l_2 must be required to exist for x' , where either $x' = x$ or x' is an ancestor of x . Observe that this form of FD does not depend on the existence of a key for the x -node. In Example 6.4.2, at node x_3 , all the following FDs of this form are generated: $\{l_3, l_1\} \rightarrow l_3$, $\{l_3, l_2\} \rightarrow l_3$, $\{l_5, l_1\} \rightarrow l_5$, $\{l_5, l_2\} \rightarrow l_5$, $\{l_5, l_3\} \rightarrow l_5$.

To keep track of the information needed to generate FDs at each x -node, we associate $att[x]$, $required[x]$, $unique[x]$, and $ancestor[x]$ with each x in $\mathbf{Rule}(\mathbf{U})$ as defined for algorithm propagation. In addition, the following variables are defined:

- $field[x]$: the set of fields in \mathbf{U} with values defined by children of x ;
- $keys[x]$: a list of sets of \mathbf{U} fields, each set mapped from a transitive key of the x -node that involves at least one attribute of x ;
- $uniqueUp[x]$: the highest ancestor y of x under which x is unique; that is, y is the variable closest to x_r such that $\Sigma \models (P(x_r, y), (P(y, x), \{\}))$;
- $descKeys[x]$: a set of keys, each for a unique keyed descendants of x ;
- $uniqueDesc[x]$: a set of variables that are unique descendants of x ;
- $uniqueAnc[x]$: the set of fields populated with unique descendants of ancestors of x .

Algorithm. Using this notation, Algorithm `minimumCover` is shown in Figure 6.10. Conceptually, the algorithm can be divided in three parts: initialization step, computation of keys, and FD generation. The initialization step is done by Algorithm `minimumCover` in

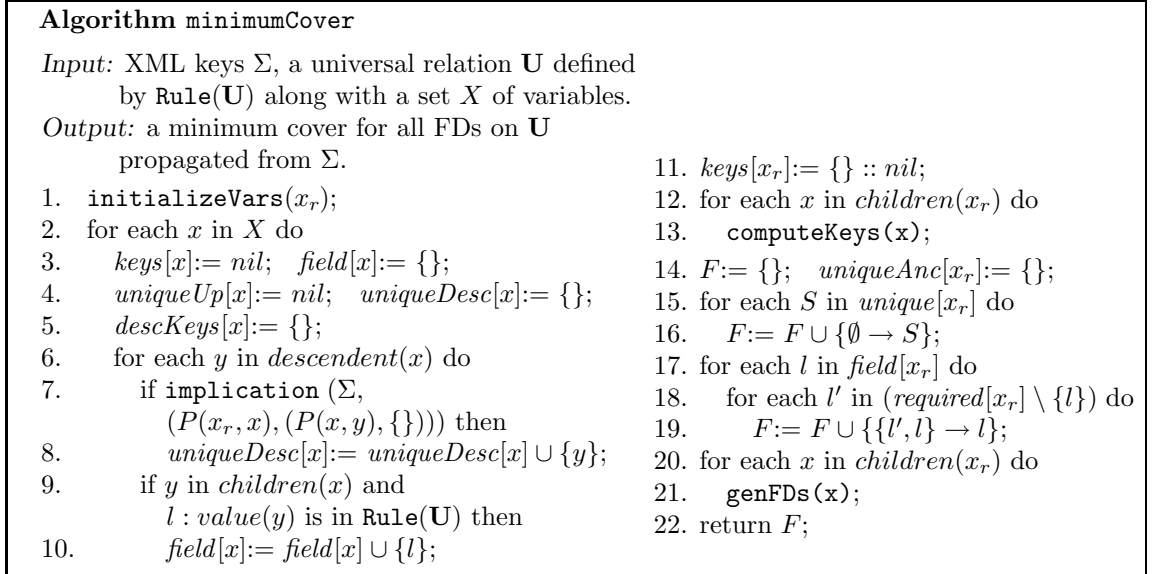


Figure 6.10: Computing minimum cover

Lines 1 to 10. First, the computation of $att[x]$, $required[x]$, $unique[x]$, and $ancestor[x]$ is executed by invoking procedure `initializeVars` (Line 1), presented in Figure 6.7. Variables $keys[x]$, $descKeys[x]$, and $uniqueUp[x]$ are given initial values in Lines 3 to 5, and variables $uniqueDesc[x]$ and $field[x]$ are computed in Lines 6 to 10. Next, the algorithm proceeds to the computation of keys. First, $keys[x_r]$ is initialized by inserting in it the empty set (Line 11); that is, since there exists a single root node in any XML tree, the empty set is a key for x_r . Then a recursive procedure `computeKeys` is invoked to each children of the root node (Line 13). After the keys of all nodes are computed, algorithm `minimumCover` proceeds to the generation of a minimum cover F of all FDs propagated from Σ via the transformation. After inserting in F FDs that can be derived for the root node (Lines 14 to 19), it invokes a recursive procedure `genFDs` to process their children (Line 21). Detailed descriptions of procedures `computeKeys` and `genFDs` are given next.

Procedure `computeKeys`. Procedure `computeKeys(x)` is presented in Figure 6.11. Given a variable x , the procedure computes $keys[x]$, and $uniqueUp[x]$ as follows: for each $(Q, (Q', S))$ in Σ , it checks whether S is contained in $att[x]$ (Line 2) and if it is the case, it computes K , the set of fields of \mathbf{U} defined by attributes in S (Line 3). It then traverses

```

procedure computeKeys (x)
  Input: x: a variable in Rule(U).
  Output: keys[x] containing minimum keys of x.
  1. for each (Q, (Q', S)) ∈  $\Sigma$  do
  2.   if  $S \subseteq \text{att}[x]$  and  $Q' \neq \epsilon$  then
  3.      $K := \{l \mid @l' \in S,$ 
            $l : \text{value}(y) \in \text{Rule}(\mathbf{U}),$ 
            $y \leftarrow x/@l' \text{ is a variable mapping}\};$ 
  4.     traverse[x] := ancestor[x];
  5.     keyFound := false;
  6.     while not keyFound and
           traverse[x] ≠ nil do
  7.       target := head(traverse[x]);
  8.       if keys[target] ≠ nil and
           implication( $\Sigma, (P(x_r, \text{target}),$ 
            $(P(\text{target}, x), S))$ ) then
  9.         keyFound := true;
  10.        if  $K \neq \{\}$  then
  11.          if x not in uniqueDesc[target] then
  12.             $K := K \cup \text{head}(\text{keys}[\text{target}]);$ 
  13.            keys[x] :=  $K :: \text{keys}[x]$ ;
  14.            else uniqueUp[x] := target;
  15.            traverse[x] := tail(traverse[x]);
  16.        if keys[x] ≠ nil then
  17.          auxK[x] := keys[x];
  18.          while auxK[x] ≠ nil do
  19.             $K := \text{head}(\text{auxK}[x]);$ 
  20.            if there exists K' in keys[x]
               s.t.  $K' \subset K$  then
  21.              remove K from keys[x];
  22.              auxK[x] := tail(auxK[x]);
  23.            if uniqueUp[x] ≠ nil then
  24.              w := parent(x);
  25.              while  $w \neq \text{uniqueUp}[x]$  and
                 (uniqueUp[w] ≠ uniqueUp[x] or
                 (uniqueUp[w] = uniqueUp[x] and
                 keys[w] = nil)) do
  26.                w := parent(w);
  27.                unique[w] :=  $\text{unique}[w] \setminus \text{unique}[x]$ ;
  28.                descKeys[w] :=
                    $\text{descKeys}[w] \cup \{\text{head}(\text{keys}[x])\}$ ;
  29.            for each y in children(x) do
  30.              computeKeys(y);

```

Figure 6.11: Procedure `computeKeys`

the ancestor path $\text{ancestor}[x]$ of x starting from the root. It finds the first ancestor target of x that is keyed, and checks whether S is a key for the x -node relative to target using Algorithm `implication` (Line 8). If these conditions are satisfied, a transitive key K' for x is constructed by combining K with the first transitive key for the ancestor target of the x -node, which is in the list $\text{keys}[\text{target}]$ (Line 12). It increments $\text{keys}[x]$ by adding this transitive key if $K \neq \{\}$ (Line 13), that is, if it contains at least one attribute of the x -node. If $K = \{\}$ then $\text{uniqueUp}[x]$ is set to target (Line 14). That is, target is the variable closest to x_r such that x is unique under target .

Besides computing all keys of the x -node, we have to guarantee that every value in $\text{keys}[x]$ is minimum. This is done as follows. First, when searching for keys, we ignore all XML keys in Σ of the form $\phi = (Q, (\epsilon, S))$ (Line 2), since a key computed from ϕ only add fields defined by S to an existent value in $\text{keys}[x]$. Second, if $K' = K_t \cup K$ is a key to be

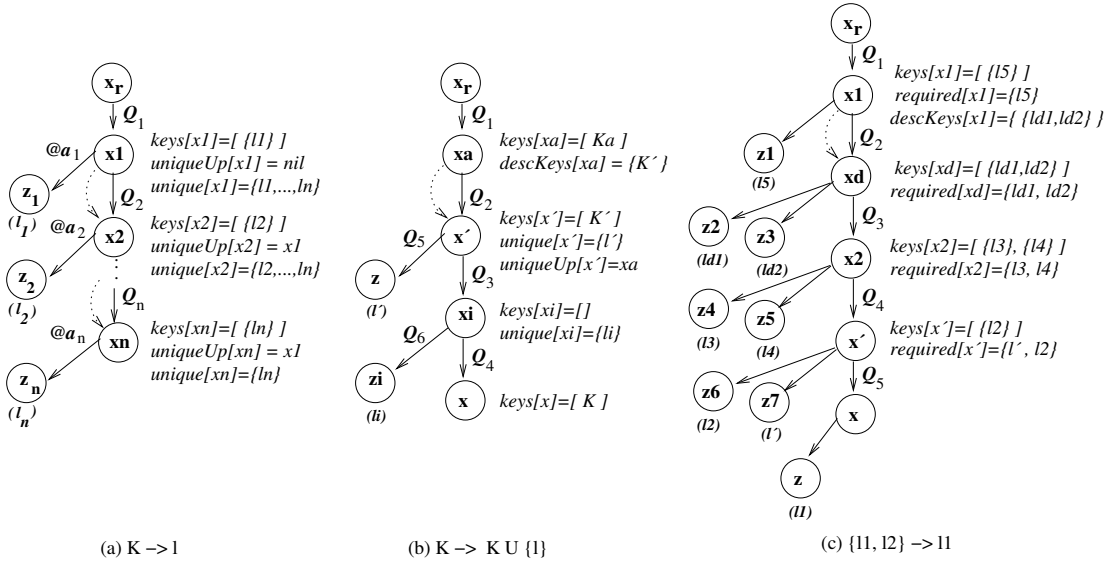


Figure 6.12: Minimization of FDs

inserted in $keys[x]$, where K_t is an element of $keys[target]$ and K is a set of fields defined by attributes of x , then K' is only inserted in F if x is not unique under $target$; that is, if x is not in $uniqueDesc[target]$ (Line 11). If this is the case, K_t is also a key for x and therefore, K' is not minimum. Third, for any two distinct keys K_1 and K_2 in $keys[x]$ such that $K_1 \subset K_2$, we remove K_2 from the list $keys[x]$ (Lines 16 to 22).

Recall that values in $keys[x]$ correspond to the left-hand side of FDs generated by the algorithm. Consider a transformation σ depicted as a table tree in Figure 6.12(a) and a set of XML keys Σ defining that each x_i has a single key $\{l_i\}$, and every x_i is unique for x_1 . Observe that for any field l_i , if $l_i \in unique[x_i]$ then l_i is also in $unique[x_j]$ for every $j < i$. That is, if l_i is unique for x_i and x_i is unique under x_1 , then x_i is unique under x_j , and l_i is unique for x_j . Therefore, for every $i > j$, $l_j \rightarrow l_i$ is an FD propagated from Σ . If all FDs of this form were generated, we would have the following set V of non-trivial FDs: $\{l_1 \rightarrow l_2, \dots, l_1 \rightarrow l_n, l_2 \rightarrow l_3, \dots, l_2 \rightarrow l_n, \dots, l_{n-1} \rightarrow l_n\}$. Observe that if $l_1 \rightarrow l_2$ and $l_2 \rightarrow l_3$ then $l_1 \rightarrow l_3$ by the transitivity rule for FD implication. That is, the set contains redundant FDs. In fact, the following is a minimum cover for V : $\{l_1 \rightarrow l_2, l_2 \rightarrow l_3, \dots, l_{n-1} \rightarrow l_n\}$. To prevent the generation of redundant FDs, the sets $unique[x_i]$ and $descKeys[x_i]$, the set that contains keys of unique descendents of x_i , are adjusted in the following way: $unique[x_{i+1}]$ is removed from $unique[x_i]$, and a key of x_{i+1}

is inserted in $descKeys[x_i]$ (Lines 23 to 28). In the above example, after this operation, $unique[x_i] = \{\}$, and $descKeys[x_i] = \{l_{i+1}\}$ for all $i \in [1, n - 1]$, and $unique[x_n] = \{l_n\}$.

Procedure genFDs. Given that procedure `computeKeys` has already adjusted sets $unique[x]$ and $descKeys[x]$, procedure `genFDs(x)` first chooses a key $K \in keys[x]$ (Line 2), and expands the minimum cover F with FDs $K \rightarrow l$ for every $l \in unique[x]$ (Lines 3 to 5), and $K \rightarrow S$ for every $S \in descKeys[x]$ (Lines 6 to 7). That is, the transitive key of the x -node determines the unique descendants of x .

In addition, to guarantee the equivalence of all transitive keys in $keys[x]$, we should expand F with $K \rightarrow K'$ and $K' \rightarrow K$ for each K' in $keys[x]$. Moreover, the following FDs should also be inserted in F to cope with the `null` value restriction: 1) $K \rightarrow K \cup \{l\}$ for every $l \in unique[x']$, where x' is an ancestor of x ; and 2) $\{l_1, l_2\} \rightarrow l_1$, where $l_1 \in fields[x]$, $l_2 \in required[x']$, and x' is either x or an ancestor of x . Although the set of FDs described generates a cover of all FDs propagated from Σ , it is not a minimum cover. In order to obtain a minimum cover of F^+ , some extra checking and bookkeeping must be performed. In the following we will describe how each of these forms of FDs are minimized.

Minimization of FDs of the form $K \rightarrow K \cup \{l\}$. Consider an FD $f = K \rightarrow K \cup \{l'\}$ where $l' \in unique[x']$, and x' is an ancestor of x as illustrated in Figure 6.12(b). Observe that if x' has a key K' , F contains $K' \rightarrow l'$ for every $l' \in unique[x']$. Therefore, we can use the following approach to derive f : we insert $K \rightarrow K \cup K'$ in F , and then obtain f by transitivity of the following FDs, derived by the augmentation rule for FD inference: $K \rightarrow K \cup K' \rightarrow K \cup \{l'\}$. More specifically, given a key K of x , because of the tree structure of an XML document, K is a key for every ancestor of x , provided that a node that corresponds to x exists. Therefore, for every x , we insert in F an FD of the form $K \rightarrow K \cup K'$, where K' is a key of the closest keyed ancestor of x (Lines 22 to 23). Then, by transitivity, we can derive $K \rightarrow K \cup K''$, where K'' is an arbitrary ancestor of x .

The only case in which a different ancestor of x should be chosen to generate an FD of this form is when its closest ancestor x' is unique under some x_a with key K_a . That is, $uniqueUp[x'] \neq nil$. If this is the case, since from F we can derive $K_a \rightarrow K'$, and $K \rightarrow K \cup K_a$, by transitivity we can obtain $K \rightarrow K \cup K'$. Therefore, we should look

```

procedure genFDs (x)
  Input: x: a variable in Rule(U).
  Output: a minimum cover of  $F^+$ .
  1. if keys[x] ≠ nil then
  2.    $K := \text{head}(\text{keys}[x]);$ 
  3.   for each l in unique[x] do
  4.     if l not in K then
  5.        $F := F \cup \{K \rightarrow l\};$ 
  6.   for each S in descKeys[x] do
  7.      $F := F \cup \{K \rightarrow (S \setminus K)\};$ 
  8.    $w := \text{parent}(x);$ 
  9.   while (keys[w] = nil) do
 10.     $w := \text{parent}(w);$ 
 11.   uniqueAnc[x] := uniqueAnc[w] ∪ unique[w];
 12.   keyedAnc := w;
 13.    $w := \text{parent}(x);$ 
 14.   while w ≠ keyedAnc do
 15.     for each l in unique[w] do
 16.       if l not in uniqueAnc[x] then
 17.          $F := F \cup \{K \rightarrow (K \cup \{l\})\};$ 
 18.         uniqueAnc[x] := uniqueAnc[x] ∪ unique[w];
 19.          $w := \text{parent}(w);$ 
 20.   while uniqueUp[w] ≠ nil or keys[w] = nil do
 21.      $w := \text{parent}(w);$ 
 22.   if head(keys[w] ) ⊄ K then
 23.      $F := F \cup \{K \rightarrow (K \cup \text{head}(\text{keys}[w]))\};$ 
 24.   for each l in field[x] do
 25.     for each l' in (required[x] \ {l}) do
 26.       if (l not in unique[x] ) or
 27.         ({l'} not in keys[x] ) then
 28.          $F := F \cup \{\{l, l'\} \rightarrow \{l\}\};$ 
 29.    $w := x;$ 
 30.   while w ≠  $x_r$  do
 31.      $w := \text{parent}(w);$ 
 32.     if required[w] ≠ {} then
 33.       allReqAreKeys := true;
 34.       for each l' in required[w] do
 35.         if {l'} not in keys[w] then
 36.           allReqAreKeys := false;
 37.         for each l in field[x] do
 38.           if ({l, l'} not in keys[x] ) or
 39.             (required[x] = {l}) then
 40.              $F := F \cup \{\{l, l'\} \rightarrow \{l\}\};$ 
 41.           if allReqAreKeys and
 42.             descKeys[w] = {} then
 43.              $l' := \text{elementOf}(\text{required}[w]);$ 
 44.             for each l in field[x] do
 45.               if (l not in unique[w] ) and
 46.                 (({l, l'} not in keys[x] ) or
 47.                  (required[x] = {l})) then
 48.                  $F := F \cup \{\{l, l'\} \rightarrow \{l\}\};$ 
 49.   auxK[x] := tail(keys[x]);
 50.   while auxK[x] ≠ nil do
 51.      $K' := \text{head}(\text{auxK}[x]);$ 
 52.      $F := F \cup \{K' \rightarrow (K \setminus K')\};$ 
 53.     auxK[x] := tail(auxK[x]);
 54.   for each y in children(x) do
 55.     genFDs(y);

```

Figure 6.13: Procedure genFDs

for the next keyed ancestor of x (Lines 20 to 21). In the example of Figure 6.12(b), $uniqueUp[x'] = x_a$, and x_a is the closest keyed ancestor of x' . Therefore FD $K \rightarrow K \cup K_a$ is inserted in F . Given this, we can obtain FDs of the form $K \rightarrow K \cup \{l\}$ for every field l that is unique for any ancestor of x' . Thus, we only have to generate FDs of the form $K \rightarrow K \cup \{l\}$ if l is a unique field for an x_i -node in the path from x to its closest keyed ancestor, unless l is also unique for an ancestor of x above x' , that is, $l \in uniqueAnc[x]$ (Lines 13 to 19).

Minimization of FDs of the form $\{l_1, l_2\} \rightarrow l_1$. FDs of the form $\{l_1, l_2\} \rightarrow l_1$ are generated at an x -node if l_1 is in $fields[x]$ and l_2 is in $required[x']$, where x' is either x or an ancestor of x . Observe that this form of FD is the only one that does not depend on the existence of a transitive key for x . First, suppose that l_2 is a required field for x . Then the FD is inserted in F unless $\{l_2\}$ is in $key[x]$, and l_1 is in $unique[x]$ (Lines 24 to 28). If this is the case, $F \models l_2 \rightarrow l_1$, and the FD can be derived by the augmentation rule for FD inference.

Now consider that l_2 is required for some ancestor x' of x as illustrated in Figure 6.12(c). Similar to the previous case, if l_2 is a key for x' , and there exists a field $l' \in required[x']$ such that $l' \neq l_2$, then $F \models l_2 \rightarrow l'$, since $required[x'] \subseteq unique[x']$, and $F \models \{l_1, l'\} \rightarrow l_1$. Thus, we can obtain $\{l_1, l_2\} \rightarrow l_1$ by the augmentation and transitivity rule for FD inference. Therefore, we should not insert an FD in F if $\{l_2\}$ is in $keys[x']$ (Lines 34 to 39). The only exception to this rule is when there exists no field $l' \in required[x']$ that satisfies the conditions above. This can happen if for every $l' \in required[x']$, l' is a key for x' as it is the case for variable x_2 of Figure 6.12(c). According to the rule above, FDs $\{l_1, l_3\} \rightarrow l_1$ and $\{l_1, l_4\} \rightarrow l_1$ are not generated, and therefore we cannot derive these FDs as described above. Given that every key for x_2 is equivalent, it suffices to choose l' to be either l_3 or l_4 and insert $\{l', l_1\} \rightarrow l_1$ in F (Lines 40 to 44). Then the FD of this form for the other field in $required[x_2]$ can be obtained by transitivity.

The only case in which such an FD involving a key of an ancestor does not have to be generated is when it has a unique keyed descendent as illustrated by variable x_1 of Figure 6.12(c). That is, $descKeys[x_1] \neq \{\}$ (Line 40). In the example, x_1 has a key $\{l_5\}$, and a

descendent x_d with a key $\{ld_1, ld_2\}$. Since x_d is unique for x_1 , $F \models l_5 \rightarrow \{ld_1, ld_2\}$. Given that both ld_1 and ld_2 are required fields for x_d , $F \models \{l_1, ld_1\} \rightarrow l_1$ and $F \models \{l_1, ld_2\} \rightarrow l_1$. Therefore, by the augmentation and transitivity rules for FD inference, $F \models \{l_1, l_5\} \rightarrow l_1$. Thus, this FD does not have to be inserted in F .

The only case that remains to be discussed is when $\{l_2, l_1\}$ is in $keys[x]$. Similar to the previous case, the FD should not be included in F if there exists a field $l' \in required[x]$ such that $l' \neq l_1$. If this is the case, $F \models \{l', l_1\} \rightarrow l_1$, and $F \models \{l_2, l_1\} \rightarrow l'$, since $l' \in unique[x]$. Thus, by transitivity, $\{l_2, l_1\} \rightarrow l_1$. Given this, $\{l_2, l_1\} \rightarrow l_1$ should be included in F only if $required[x] = \{l_1\}$ (Line 38 and Line 43), that is, if a field $l' \neq l_1$ that satisfies the above conditions does not exist.

Minimization of FDs of the form $K \rightarrow K'$. Let K be a key in $keys[x]$. We will first show that if we generate all forms of FDs described above for K , then for any other key K' in $keys[x]$, $F \models K \rightarrow K'$. Let $K' = (L \cup S)$, where for every field $l \in L$, l is in $required[x]$, and for every field $l' \in S$, l' is in $required[y]$, where y is an ancestor of x . Since a required field is also unique, $K \rightarrow l$ for every $l \in L$, and $K \rightarrow K \cup \{l'\}$ for every $l' \in S$. Therefore, by the augmentation and transitivity rules for FD inference, $K \rightarrow K \cup K'$. Observe that K' must contain at least one field l_1 in $required[x]$. Therefore, for every label $l_2 \in K$, $\{l_2, l_1\} \rightarrow l_1$. Thus, $K \cup K' \rightarrow K'$, and by transitivity, $K \rightarrow K'$. Therefore, to maintain the equivalence among all keys in $keys[x]$, we only have to choose one $K \in keys[x]$ to generate the FDs described, and then expand F with $K' \rightarrow K$ for the other keys in the list (Lines 45 to 49).

Example 6.15 Given transformation σ and the set Σ of XML keys of Example 6.2.2, Algorithm `minimumCover` returns the FDs given in Example 6.2.2. Specifically, the algorithm traverses the table tree of Figure 6.5 top-down starting at the root. At node x_b , two FDs are generated: `bookIsbn` \rightarrow `bookTitle` and `bookIsbn` \rightarrow `authContact`. Here $keys[x_b] = [\{\text{bookIsbn}\}]$. At node y_c , FD `bookIsbn, chapNum` \rightarrow `chapName` is included in F since $keys[y_c]$ is $[\{\text{bookIsbn, chapNum}\}]$, which is constructed by combining the field name populated with `@number`, a key of y_c relative to x_b , and the key in $keys[x_b]$. Similarly, at node z_s FD `bookIsbn, chapNum, secNum` \rightarrow `secName` is inserted into F . The minimum

cover for all the FDs propagated from Σ via σ also includes the following FDs of the form $\{l_1, l_2\} \rightarrow l_1$:

$$\begin{array}{ll}
\{bookTitle, bookIsbn\} \rightarrow bookTitle & \{chapNum, bookIsbn\} \rightarrow chapNum \\
\{chapName, chapNum\} \rightarrow chapName & \{chapName, bookIsbn\} \rightarrow chapName \\
\{bookAuthor, bookIsbn\} \rightarrow bookAuthor & \{authContact, bookIsbn\} \rightarrow authContact \\
\{secNum, chapNum\} \rightarrow secNum & \{secNum, bookIsbn\} \rightarrow secNum \\
\{secName, secNum\} \rightarrow secName & \{secName, chapNum\} \rightarrow secName \\
\{secName, bookIsbn\} \rightarrow secName &
\end{array}$$

□

One can show that the size of $keys[x]$ is quadratic in the size of Σ , $unique[x]$ and $required[x]$ are bounded by the size of $\mathbf{Rule}(\mathbf{U})$, and the set X is no larger than the size of $\mathbf{Rule}(\mathbf{U})$; thus the set F is bounded by m^2n^3 , where m and n are the sizes of XML keys Σ and table tree T_R , respectively. That is, the size (thus the cardinality) of F is at most $O(m^2n^3)$, a polynomial in the input size.

The complexity of the algorithm is $O(m^4n^4)$ time. It is not hard to see that for each variable x in $\mathbf{Rule}(\mathbf{U})$, Procedure $\mathbf{computeKeys}(x)$ takes $O(m^4n^3)$ time. Indeed, for each key in Σ (Line 1) and for each ancestor $target$ of x (Line 6), it checks implication of XML keys using Algorithm $\mathbf{implication}$ (Line 8), which takes $O(m^2n^2)$ time since the key to be checked is no larger than the size of $\mathbf{Rule}(\mathbf{U})$. The minimization of $keys[x]$ (Lines 16 to 22) takes at most $O(m^4)$ time because there exists at most m values in $keys[x]$ each of size at most m . The last step in the procedure (Lines 23 to 28) takes no more than $O(n)$ time. Therefore, the overall cost of procedure $\mathbf{computeKeys}$ for a variable x is $O(m^4n^3)$.

Before analyzing procedure $\mathbf{genFDs}(x)$, observe that the size of $unique[x]$, $uniqueAnc[x]$, $required[x]$, and $field[x]$ is at most $O(n)$. Since one key K is selected to generate FDs of the form $K \rightarrow l$ and $K \rightarrow S$, where $l \in unique[x]$, and $S \in unique[x]$, Lines 2 to 7 take $O(n)$ time. The generation of FDs of the form $K \rightarrow (K \cup S)$ takes $O(n^3)$ time (Lines 8 to 23), and the generation of FDs of the form $\{l, l'\} \rightarrow l$ for required fields for x takes $O(n^2)$ time (Lines 24 to 28). Generating this form of FDs for ancestors of x (Lines 29 to 44) takes $O(m^2n^3)$ time given that the outer loop is executed at most n times, each for an

ancestor of x . Each ancestor w has at most n fields in $required[w]$, x has at most n fields in $fields[x]$, and for each combination it is tested if the value is in $keys[x]$ of size $O(m^2)$. The second loop (Lines 40 to 44) takes at most the same time. The generation of FDs of the form $K \rightarrow K'$ for each K' in $keys[x]$ takes $O(m)$ time (Lines 45 to 49), since there exists at most $O(m)$ values in $keys[x]$. Therefore, the overall cost of procedure `genFDs` for a variable x is $O(m^2n^3)$.

Since Procedure `computeKeys` and `genFDs` processes each variable x in $\text{Rule}(\mathbf{U})$ once, the total time for executing both procedures (Lines 12 to 21) is at most $O(m^4n^4)$ time. The initialization steps (Lines 1 to 10) take at most $O(m^2n^4)$ time, and therefore, algorithm `minimumCover` costs at most $O(m^4n^4)$. Since Σ and $\text{Rule}(\mathbf{U})$ are usually small, this algorithm is efficient in practice. The experimental results of Section 6.5 also show that it substantially outperforms Algorithm `naive`.

A final remark is that, although one can generalize Algorithm `minimumCover` to check XML key propagation instead of using Algorithm `propagation`, there are good reasons for not doing so. The complexity of Algorithm `minimumCover` is higher than that of Algorithm `propagation` ($O(m^4n^4)$ vs. $O(m^2n^4)$). In short, Algorithm `propagation` is best used to inspect a predefined relational schema, whereas Algorithm `minimumCover` helps normalize a universal relation at the early stage of relational design.

6.4.3 The Correctness of the Minimum Cover Algorithm

We now prove the correctness of our algorithm for computing a minimum cover:

Theorem 6.4 *Given a set Σ of XML keys and a universal relation \mathbf{U} defined by a transformation σ , Algorithm `minimumCover` computes a minimum cover for all the FDs on \mathbf{U} propagated from Σ via σ .*

To prove the theorem, we will first show that the set F resulting from the algorithm indeed generates all FDs necessary to cope with the `null` value restriction. More specifically, in Lemma 6.5 we will show that for any field $l_1 \in fields[x]$, and any field l_2 required for x ,

$F \models \{l_1, l_2\} \rightarrow l_1$. Then, in Lemma 6.6, we will show that if $K = \text{head}(\text{keys}[x])$ then $F \models K \rightarrow K \cup \{l\}$ for every field l that is unique for an ancestor of x , and moreover, $F \models K \cup \{l\} \rightarrow K$ for every required field l for x .

Based on these two lemmas we develop the proof of the theorem in three parts. The first is to show that for any FD $\phi = Y \rightarrow Z$ in F it is indeed the case that $\Sigma \models_{\sigma} \phi$. The proof is based on the idea that Y corresponds to the set of fields defined by a transitive set of keys Θ for an x -node, and fields in Z are unique for x . Given that $\Sigma \models \Theta$, for any XML tree T if $T \models \Sigma$ then there exists no two nodes in T agreeing on Y . Since fields in Z are unique, $\sigma(T) \models \phi$.

The second part is to show that all FDs propagated from Σ can be derived from F . That is, if $\Sigma \models_{\sigma} \phi$ then $F \models \phi$. Similar to the correctness proof of algorithm `propagation` presented in Section 6.3.2, we decompose the FD $\phi = Y \rightarrow Z$ to a set of simple FDs $\phi_i : Y_i \rightarrow Z_i$ such that labels in $Y_i \cup Z_i$ are defined along a path from the root x_r to a variable x_i . Given this, we encode Y_i to a set of transitive keys Θ and Z_i to a set of keys asserting the uniqueness of each field in Z_i such that $\Sigma \models_{\sigma} Y_i \rightarrow Z_i$ if and only if $\Sigma \models \Theta \cup \Gamma$. Then, in Lemma 6.7, we show that $F \models Y_i \rightarrow \text{head}(\text{keys}[x_i])$ and $F \models \text{head}(\text{keys}[x_i]) \rightarrow Y_i$; that is, they are equivalent according to F , and moreover, $F \models \text{head}(\text{keys}[x_i]) \rightarrow Z_i$. Thus, by transitivity, we derive $F \models Y_i \rightarrow Z_i$. Given that for every ϕ_i the implication holds, we can obtain $F \models Y \rightarrow Z$ by the augmentation and transitivity rules for FD inference.

The third part is to show that F is not only a cover of all FDs propagated from Σ , but a minimum cover. That is, F does not contain any redundant FDs. We will show that for each of the forms of FDs generated by the algorithm, $F \setminus \{\phi\} \not\models \phi$.

Lemma 6.5 *Let x be a variable in $\text{Rule}(R)$, and l a field defined by a child of x . For every label l' , if l' is required for x then $F \models \{l, l'\} \rightarrow l$.*

Proof. Let l be defined by $l : \text{value}(y)$, where $y \leftarrow x/Q$, and l' be defined by $l' : \text{value}(y')$, where $y' \leftarrow x'/@a$, and either $x' = x$ or x' is an ancestor of x . We will show by induction on $|P(x', x)|$ that for all fields in $\text{required}[x']$, $F \models \{l, l'\} \rightarrow l$. For the base case, $|P(x', x)| = 0$, that is, $x = x'$. This case is direct because procedure `genFDs` inserts FD $\{l, l'\} \rightarrow l$ for all

$l \in \text{field}[x]$, and all $l' \in \text{required}[x]$, except when $\{l'\}$ is in $\text{keys}[x]$, and l is in $\text{unique}[x]$. But then $F \models l' \rightarrow l$, and the FD can be obtained by augmentation of l .

Suppose the statement holds for all descendants of x' . We have to show that it also holds for x' . Let l' be an arbitrary field in $\text{required}[x']$. Observe that, by construction, if $\{l'\}$ is not in $\text{keys}[x']$ and $\{l, l'\}$ is not in $\text{keys}[x]$ then F contains $\{l, l'\} \rightarrow l$. Therefore, we have to show that in these two cases the FD can be derived from F . First, consider that $\{l, l'\}$ is in $\text{keys}[x]$. If there exists a field $l'' \neq l$ in $\text{required}[x]$ then, by construction, F contains both $\{l, l''\} \rightarrow l$, and $\{l, l'\} \rightarrow l''$, since $\text{required}[x] \subseteq \text{unique}[x]$. Therefore, by the augmentation and transitivity rules for FD inference, $F \models \{l, l'\} \rightarrow l$. If such an l'' does not exist, then by construction the FD is inserted in F and therefore the implication also holds.

Now consider that $\{l'\}$ is in $\text{keys}[x']$. Similar to the previous case, if there exists an $l'' \neq l'$ in $\text{required}[x']$ such that $F \models \{l, l''\} \rightarrow l$ then $\{l, l'\} \rightarrow l$ can be obtained by the augmentation and transitivity rules for FD inference, since $F \models l' \rightarrow l''$. Such an l'' does not exist if for all fields $l_i \in \text{required}[x']$, $\{l_i\}$ is a key for x' . Let $\text{required}[x'] = \{l_1, \dots, l_k\}$. It suffices to show that for some l_i , $F \models \{l_i, l\} \rightarrow l$. For if it holds, since for all l_j , $j \in [1, k]$, $F \models \text{head}(\text{keys}[x']) \rightarrow l_j$, and $F \models l_j \rightarrow \text{head}(\text{keys}[x'])$, we can obtain $F \models \{l_j, l\} \rightarrow l$ for any l_j by the augmentation and transitivity rules for FD inference. By construction, an FD $\{l_i, l\} \rightarrow l$ is inserted in F , except when $\text{descKeys}[x'] \neq \{\}$. If this is the case, then F contains $\text{head}(\text{keys}[x']) \rightarrow \text{head}(\text{keys}[z])$, where $\text{head}(\text{keys}[z]) \in \text{descKeys}[x']$. By inductive hypothesis, for every field $l_z \in \text{head}(\text{keys}[z])$, $F \models \{l, l_z\} \rightarrow l$. Therefore, $F \models \text{head}(\text{keys}[z]) \cup \{l\} \rightarrow l$. Thus, by the augmentation and transitivity rules, $F \models \text{head}(\text{keys}[x']) \cup \{l\} \rightarrow l$. That is, for some $l_i \in \text{required}[x']$, $\{l, l_i\} \rightarrow l$ is in F , as we wanted to show. \square

Lemma 6.6 *Let x be a keyed variable, and $K = \text{head}(\text{keys}[x])$. For every unique label l for x' , where either x' is an ancestor of x or $x' = x$, $F \models K \rightarrow K \cup \{l\}$. For every required label l for x , $F \models K \cup \{l\} \rightarrow K$ if $x \neq x_r$.*

Proof. We will first show that for every required label l for x $F \models K \cup \{l\} \rightarrow K$ if $x \neq x_r$. By construction, there exists a label $l' \in K$ such that $l' \in \text{fields}[x]$. Therefore, by Lemma 6.5, since l is required for x , $F \models \{l, l'\} \rightarrow l'$. Therefore, by augmenting both sides with K , we obtain $K \cup \{l\} \rightarrow K$.

A roadmap of the proof that $F \models K \rightarrow K \cup \{l\}$ is as follows. First, we will show that for every keyed ancestor w of x , $F \models K \rightarrow K \cup \text{head}(\text{keys}[w])$. Given this, let l be defined by $l : \text{value}(x_l)$ and w be either x or a common keyed ancestor of x_l and x . We will then show that $F \models \text{head}(\text{keys}[w]) \rightarrow \text{head}(\text{keys}[w]) \cup \{l\}$. From these two FDs we can derive $K \rightarrow K \cup \text{head}(\text{keys}[w]) \cup \{l\}$ by the augmentation and transitivity rules for FD inference. Since every field $l_w \in \text{head}(\text{keys}[w])$ is required for x , $F \models K \cup \{l_w\} \rightarrow K$, and thus $F \models K \cup \text{head}(\text{keys}[w]) \rightarrow K$. Applying again the augmentation and transitivity rules, we finally obtain $K \rightarrow K \cup \{l\}$.

To show that for every keyed ancestor w of x , $F \models K \rightarrow K \cup \text{head}(\text{keys}[w])$, observe that if $\text{uniqueUp}[w] = w'$, then by construction, w' is keyed and $F \models \text{head}(\text{keys}[w']) \rightarrow \text{head}(\text{keys}[w])$. Therefore, it suffices to show that for every keyed ancestor w of x with $\text{uniqueUp}[w] = \text{nil}$, $F \models K \rightarrow K \cup \text{head}(\text{keys}[w])$. The proof is by induction on the number of keyed variables w' between x and w with $\text{uniqueUp}[w'] = \text{nil}$. For the base case, w is the closest keyed ancestor of x with $\text{uniqueUp}[w] = \text{nil}$, and it is direct because F contains $K \rightarrow K \cup \text{head}(\text{keys}[w])$.

Now suppose that w is the n -th keyed ancestor of x with $\text{uniqueUp}[w] = \text{nil}$, and the statement holds for w and descendants of w . We have to show that it holds for w' , its closest keyed ancestor. Let $H_w = \text{head}(\text{keys}[w])$, and $H_{w'} = \text{head}(\text{keys}[w'])$. By inductive hypothesis, the following FDs can be inferred from F :

$$\begin{aligned} \phi_1 : K &\rightarrow K \cup H_w \\ \phi_2 : H_w &\rightarrow H_w \cup H_{w'}. \end{aligned}$$

Since every $l \in H_w$ is required for x , $F \models K \cup \{l\} \rightarrow K$. Therefore, the following FD can also be inferred from F :

$$\phi_3 : K \cup H_w \rightarrow K$$

We can now obtain $K \rightarrow K \cup H_{w'}$ by transitivity of the following FDs:

$$\begin{array}{ll} K \rightarrow K \cup H_w & \phi_1 \\ K \cup H_w \rightarrow K \cup H_{w'} \cup H_w & \text{augmentation of } \phi_2 \text{ with } K \\ K \cup H_w \cup H_{w'} \rightarrow K \cup H_{w'} & \text{augmentation of } \phi_3 \text{ with } H_{w'}. \end{array}$$

Given that for every keyed ancestor w of x , $F \models K \rightarrow K \cup \text{head}(\text{keys}[w])$, to complete the proof, we have to show that $F \models \text{head}(\text{keys}[w]) \rightarrow \text{head}(\text{keys}[w]) \cup \{l\}$ for every required field l for x , where l is defined by a variable x_l , and w is either x or a common keyed ancestor of x_l and x . We have two cases to consider.

Case 1: the closest common ancestor w of x_l and x is keyed. Let l be a field in $\text{unique}[w]$. If $l \notin \text{head}(\text{keys}[w])$ then procedure **genFDs** inserts $\text{head}(\text{keys}[w]) \rightarrow l$ in F . Otherwise, since every $l_w \in \text{head}(\text{keys}[w])$ is required for w , $F \models \{l_w, l\} \rightarrow l$. Therefore, by the augmentation and transitivity rules for FD inference we obtain $F \models \text{head}(\text{keys}[w]) \rightarrow l$. By applying the augmentation rule again we obtain $\text{head}(\text{keys}[w]) \rightarrow \text{head}(\text{keys}[w]) \cup \{l\}$.

Case 2: the closest common ancestor w of x_l and x is not keyed. Let x' be w closest keyed ancestor, and l a field in $\text{unique}[w]$. By construction, procedure **genFDs** inserts $\text{head}(\text{keys}[x]) \rightarrow \text{head}(\text{keys}[x]) \cup \{l\}$ in F , unless $l \in \text{uniqueAnc}[x']$. If this is the case, then there exists a keyed ancestor w' of x' such that $F \models \text{head}(\text{keys}[w']) \rightarrow \text{head}(\text{keys}[w']) \cup \{l\}$. Given that w' is also an ancestor of x , this completes the proof. \square

Now we are ready to show that F is a minimum cover of all FDs propagated from Σ via σ . Let F be the set of FDs computed by algorithm **minimumCover**. We have to show that if $\phi \in F$ then $\Sigma \models_{\sigma} R : \phi$ (Part 1), and if $\Sigma \models_{\sigma} R : \phi$ then $F \models \phi$ (Part 2). Moreover, we have to show that there is no subset F' of F such that $F' \models F$ (Part 3).

Proof of Part 1. Let $\phi = Y \rightarrow Z$ be an arbitrary FD in F generated by procedure **genFDs** when processing a variable y . We have to show that $\Sigma \models_{\sigma} R : \phi$. That is, we have to show that for any XML tree T such that $T \models \Sigma$, $\sigma(T) \models \phi$. By construction, ϕ can be of two forms: 1) $\{l, l'\} \rightarrow l$, and 2) $K \rightarrow Z$, where K is in $\text{keys}[y]$.

Consider FDs of the first form. Since $\{l\} \subset \{l, l'\}$ it cannot be the case that for any two tuples $t_1, t_2 \in \sigma(T)$, $\pi_{l,l'}(t_1) = \pi_{l,l'}(t_2)$ and $\pi_l(t_1) \neq \pi_l(t_2)$. Therefore, we only have to show that the FD does not violate the null restriction for FD satisfaction. This is indeed the case since by construction, if $l \in \text{field}[y]$ then $l' \in \text{required}[y']$, where either y' is y or an ancestor of y . That is, for any XML tree T that satisfies Σ , whenever there exists a node $n \in \llbracket P(x_r, y) \rrbracket$, it must be the case that there exists a node $n' \in \llbracket P(x_r, y') \rrbracket$ such that $n \in n' \llbracket P(y', y) \rrbracket$. Moreover, since $l' \in \text{required}[y']$, if $l' : \text{value}(z)$ is in $\text{Rule}(\mathbf{U})$ and $z \leftarrow y' / @a$ then $n' \llbracket P(y', @a) \rrbracket$ is not empty. Therefore, by the semantics of transformations, for any tuple $t \in \sigma(T)$, whenever $\pi_l(t) \neq \text{null}$, $\pi_{l'}(t) \neq \text{null}$. Thus, $\sigma(T)$ satisfies the null restriction for FD satisfaction.

Now consider the second form of FDs: $\phi : K \rightarrow Z$, where K is a value in $\text{keys}[y]$. Recall that K is composed of required attributes of ancestors of y , and moreover, contains at least one required attribute of y . As a consequence, the existence of a node v in $\llbracket P(x_r, y) \rrbracket$ implies that $\pi_Y(t)$ does not contain null . Therefore, we have to show that: 1) the existence of nodes to populate fields in Z guarantees the existence of a node v in $\llbracket P(x_r, y) \rrbracket$, and 2) for any node v in $\llbracket P(x_r, y) \rrbracket$, there exists at most one node in T that corresponds to each label in Z .

The first condition guarantees that whenever $\pi_Y(t)$ contains null so does $\pi_Z(t)$. It suffices to show that there exists a label $l \in Z$ such that $l \in \text{descendent}(y)$. This is indeed the case, since by construction either Z involves K , or $Z = \{l\}$, where l is a unique descendent of y , or $Z = \text{head}(\text{keys}[w])$, where w is a descendent of y , or $Z = K'$, where K' is in $\text{keys}[y]$.

The second condition guarantees that there exists no two tuples in $\sigma(T)$ that agree on Y but not on Z . By construction, K corresponds to fields populated with attributes in a transitive set of keys for y . That is, this set of attributes uniquely identifies a node in $\llbracket P(x_r, y) \rrbracket$. Let this transitive key defined by K be Θ . Since $\Sigma \models \Theta$, by Lemma 6.2, there exists no two nodes in T agreeing on K . Therefore, we only have to show that for any node v in $\llbracket P(x_r, y) \rrbracket$, there exists at most one node in T that corresponds to every label in Z . For if it holds there are no two tuples in $\sigma(T)$ that agree on the values of K , and for each value of K there exists at most one value for Z . By construction, either Z is in $\text{unique}[y]$, or

Z in $descKeys[y]$, or Z in $keys[y]$, or $Z = K \cup L$, where for every $l \in L$, l is unique for y or for some ancestor of y .

The first two cases are direct because by the computation of $unique[y]$ and $descKeys[y]$, either $Z = \{l\}$, $l : value(z')$, z' is a descendent of y , and $\Sigma \models (P(x_r, y), (P(y, z'), \{\}))$, or $Z = head(keys[w])$, and $\Sigma \models (P(x_r, y), (P(y, w), \{\}))$. For the third and fourth cases, observe that any K' in $keys[y]$ is composed of only required labels of y , and by definition, every required label of y is also unique for y or for an ancestor of y . Because of the tree structure of T , for every node n in $\llbracket P(x_r, y) \rrbracket$, and every ancestor y' of y there exists a unique node $n' \in \llbracket x_r, y' \rrbracket$ such that $n \in n' \llbracket P(y', y) \rrbracket$. Therefore, by the semantics of transformations, for each value of K there exists at most one value for Z . This completes the proof that for any $\phi \in F$, $\Sigma \models_\sigma R : \phi$.

Proof of Part 2. We now have to show that if $\Sigma \models_\sigma R : \phi$ then $F \models \phi$. Let $\phi = Y \rightarrow Z$. As shown in the proof of Theorem 6.3, ϕ can be decomposed to a set of simple FDs $\phi_i = Y_i \cup Z_i$, denoted as $fdsDefBy(\phi)$. According to Lemma 6.1, $\Sigma \models_\sigma \phi$ if and only if for every i , $\Sigma \models_\sigma \phi_i$. We then encode Z_i as a set of XML keys Γ that asserts the uniqueness of each field in Z_i . Moreover, we define Y_{min} to be a minimum subset of Y_i such that $\Sigma \models_\sigma Y_{min} \rightarrow Z_i$, and encode Y_{min} as a transitive set of keys Θ for an x_i -node. By Lemma 6.3, $\Sigma \models \Theta \cup \Gamma$.

Therefore, to prove that $F \models \phi_i : Y_i \rightarrow Z_i$. we will first show that $F \models Y_{min} \rightarrow head(keys[x_i])$, and $F \models head(keys[x_i]) \rightarrow Y_{min}$. This is asserted by Lemma 6.7 given below and it is based on the definition of Θ . Given this, we then show that for every field $l \in Y_i$ but not in Y_{min} , denoted as Y_{ext} , $F \models head(keys[x_i]) \cup \{l\} \rightarrow head(keys[x_i])$. This is direct from Lemma 6.6, since every l in Y_{ext} is required for x_i . Thus, by the augmentation and transitivity rules, $F \models head(keys[x_i]) \cup Y_{ext} \rightarrow head(keys[x_i])$. Given that $F \models Y_{min} \rightarrow head(keys[x_i])$, we can obtain $Y_{min} \cup Y_{ext} \rightarrow head(keys[x_i]) \cup Y_{ext}$ by augmenting both sides with Y_{ext} . Therefore, by transitivity, $F \models Y_{min} \cup Y_{ext} \rightarrow head(keys[x_i])$.

To complete the proof, we have to show that $F \models head(keys[x_i]) \rightarrow Z_i$. Let $Z_i = Z_x \cup Z_a$, where $Z_x \subseteq unique[x_i]$, and $Z_a = Z_i \setminus Z_x$. Observe that in order to satisfy the null restriction for FD satisfaction, $Z_x \neq \emptyset$. Recall that by Lemma 6.3, $\Sigma \models \Gamma$, where $\Gamma =$

$\{(P(x_r, w), (P(w, z), \{\})) \mid z \in \text{var}(Z_i), \text{ and } w \text{ is the closest ancestor of the } z \text{ in } W_i\}$. That is, every label $l \in Z_i$ is unique for some w , where $w = x_i$, or w is an ancestor of x_i . Therefore, by Lemma 6.6, for every $l \in Z_a$, $F \models \text{head}(\text{keys}[x_i]) \rightarrow \text{head}(\text{keys}[x_i]) \cup \{l\}$. Applying the augmentation and transitivity rules we can derive $\text{head}(\text{keys}[x_i]) \rightarrow \text{head}(\text{keys}[x_i]) \cup Z_a$. By construction, $Z_x \subseteq \text{unique}[x_i]$, and procedure **genFDs** generates $\text{head}(\text{keys}[x_i]) \rightarrow l$ for every $l \in Z_x$. Therefore, $F \models \text{head}(\text{keys}[x_i]) \rightarrow Z_x$. By augmenting this FD with Z_a we obtain $\text{head}(\text{keys}[x_i]) \cup Z_a \rightarrow Z_x \cup Z_a$. Therefore, by transitivity, $\text{head}(\text{keys}[x_i]) \rightarrow Z_x \cup Z_a$, that is, $F \models \text{head}(\text{keys}[x_i]) \rightarrow Z_i$. Since $F \models Y_i \rightarrow \text{head}(\text{keys}[x_i])$, by the augmentation and transitivity rules for FD inference we derive $F \models Y_i \rightarrow Z_i$. Given that for all i , $F \models Y_i \rightarrow Z_i$, we can conclude that $F \models Y \rightarrow Z$.

Lemma 6.7 *Let $x \neq x_r$ be a variable, and Θ be a minimum transitive set of XML keys for x such that $\Sigma \models \Theta$. Let K be the set of labels defined by attributes of keys in Θ , that is, $K = \{l \mid l : \text{value}(y), y \leftarrow w/@a, @a \in S, (Q, (Q', S)) \in \Theta\}$. Then $F \models \text{head}(\text{keys}[x]) \rightarrow K$, and $F \models K \rightarrow \text{head}(\text{keys}[x])$.*

Proof. Let Θ be $\{\kappa_1, \dots, \kappa_n\}$, where each κ_i is defined as follows. Let $V = \{y_1, \dots, y_n\}$ be the subset of variables along the path $P(x_r, x)$ with children that populate fields in K . Define $\{Y_1, \dots, Y_n\}$ to be the subsets of K such that each Y_i correspond to fields populated by children of y_i , and K_i to be the set $\{Y_j \mid j \leq i\}$. Given this, we define $\kappa_i = (P(x_r, y_{i-1}), (P(y_{i-1}, y_i), \text{Kattr}(y_i)))$, where $\text{Kattr}(y_i)$ denotes the set of attributes of y_i that define fields in Y_i .

We will show by induction on n , the cardinality of Θ , that for all $i \in [1, n]$, $F \models \text{head}(\text{keys}[y_i]) \rightarrow K_i$ and $F \models K_i \rightarrow \text{head}(\text{keys}[y_i])$. For the base case, $n = 1$, i.e. $\Theta = \{(\epsilon, (Q, \text{Kattr}(y_1)))\}$. Since $\text{Kattr}(y_1)$ is minimum, there must exist a key $(\epsilon, (Q', \text{Kattr}(y_1)))$ in Σ , where $Q \subseteq Q'$. Therefore, algorithm **computeKeys** inserts Y_1 in $\text{keys}[y_1]$. Since $Y_1 \subseteq \text{required}[x] \subseteq \text{unique}[x]$, algorithm **genFDs** generates $\text{head}(\text{keys}[x]) \rightarrow l$ for every $l \in Y_1$. Therefore, by the augmentation and transitivity rules for FD inference, $F \models \text{head}(\text{keys}[x]) \rightarrow K$. The other direction is direct because an FD $K \rightarrow \text{head}(\text{keys}[x]) \setminus K$ is also generated by **genFDs** and inserted in F .

Assume the statement holds for every $i < n$. We will show that it holds for n . Observe that when processing variable y_n , procedure `computeKeys` inspects all keys ϕ in Σ to check whether ϕ can be part of a transitive set of keys for the y_n -node. Since Θ is minimum, if $(Q_k, (Q'_k, Kattr(y_n))) \in \Theta$, then there exists a key $\phi = (Q, (Q', Kattr(y_n)))$ in Σ . Therefore, starting from x_r , `computeKeys` checks for each keyed ancestor y_a of y_n whether $Kattr(y_n)$ is a key for the y_n -node with respect to y_a . That is, if $\Sigma \models (P(x_r, y_a), (P(y_a, y_n), Kattr(y_n)))$. First, suppose that $y_a = y_{n-1}$, and let $H_{n-1} = head(keys[y_{n-1}])$. In this case, an element is inserted in $keys[y_n]$ by combining H_{n-1} with Y_n . By inductive hypothesis, $F \models K_{n-1} \rightarrow H_{n-1}$, and procedure `genFDs` inserts in F an FD $(H_{n-1} \cup Y_n) \rightarrow head(keys[y_n])$. Therefore, by the augmentation and transitivity rules, $F \models K_{n-1} \cup Y_n \rightarrow head(keys[y_n])$. To show the other direction, observe that since every $l \in Y_n$ is unique for y_n , F contains $head(keys[y_n]) \rightarrow l$. Therefore, $F \models \phi'_1$, where $\phi'_1 = head(keys[y_n]) \rightarrow Y_n$. By inductive hypothesis, $F \models \phi'_2$, where $\phi'_2 = H_{n-1} \rightarrow K_{n-1}$. Therefore, we can obtain $head(keys[y_n]) \cup H_{n-1} \rightarrow K_{n-1} \cup Y_n$ by transitivity of the following FDs:

$$\begin{array}{ll} head(keys[y_n]) \cup H_{n-1} \rightarrow Y_n \cup H_{n-1} & \text{augmentation of } \phi'_1 \text{ with } H_{n-1} \\ H_{n-1} \cup Y_n \rightarrow K_{n-1} \cup Y_n & \text{augmentation of } \phi'_2 \text{ with } Y_n \end{array}$$

Since every label $l \in H_{n-1}$ is unique for y_{n-1} , and y_{n-1} is an ancestor of x , by Lemma 6.6, $F \models head(keys[y_n]) \rightarrow head(keys[y_n]) \cup \{l\}$. Therefore, by the augmentation and transitivity rules, $F \models head(keys[y_n]) \rightarrow head(keys[y_n]) \cup H_{n-1}$. Therefore, by transitivity, $F \models head(keys[y_n]) \rightarrow K_{n-1} \cup Y_n$.

Now, suppose that $P(x_r, y_{n-1}) = P(x_r, y_a)/Q$. That is, $Kattr(y_n)$ is a key for the y_n -node with respect to y_a and y_a is a keyed ancestor of y_{n-1} . Let $H_a = head(keys[y_a])$. In this case, a value is inserted in $keys[y_n]$ by combining H_a with Y_n . Thus, according to the previous discussion, the following FDs are consequences of F :

$$\begin{array}{l} \phi_1 : H_a \cup Y_n \rightarrow head(keys[y_n]) \\ \phi_2 : head(keys[y_n]) \rightarrow H_a \cup Y_n \end{array}$$

We have to show that $F \models H_{n-1} \cup Y_n \rightarrow H_a \cup Y_n$ and $F \models H_a \cup Y_n \rightarrow H_{n-1} \cup Y_n$.

Before doing so, we will first establish the implication of some FDs involving H_{n-1} and H_a . Observe that since y_a is an ancestor of y_{n-1} , every $l \in H_a$ is required for y_{n-1} , and unique for y_a . Therefore, by Lemma 6.6 the following FDs are implied by F :

$$\begin{aligned}\phi_3 &: H_a \cup H_{n-1} \rightarrow H_{n-1} \\ \phi_4 &: H_{n-1} \rightarrow H_a \cup H_{n-1}.\end{aligned}$$

By inductive hypothesis the following FDs are also consequences of F :

$$\begin{aligned}\phi_5 &: \text{head}(\text{keys}[y_{n-2}] \cup Y_{n-1}) \rightarrow H_{n-1} \\ \phi_6 &: H_{n-1} \rightarrow \text{head}(\text{keys}[y_{n-2}]) \cup Y_{n-1}.\end{aligned}$$

By construction, $Y_{n-1} \subseteq \text{unique}[y_{n-1}]$. Therefore, F implies the following FD:

$$\phi_7 : H_{n-1} \rightarrow Y_{n-1}.$$

From these FDs we can infer the following:

$$\begin{aligned}\phi_8 &: H_{n-1} \cup H_a \rightarrow Y_{n-1} \cup H_a \\ \phi_9 &: Y_{n-1} \cup H_a \rightarrow H_{n-1} \cup H_a.\end{aligned}$$

ϕ_8 is obtained by augmenting ϕ_7 with H_a , while ϕ_9 is obtained by transitivity of the following FDs:

$$\begin{array}{ll} H_a \rightarrow H_a \cup \text{head}(\text{keys}[y_{n-2}]) & \text{by Lemma 6.6} \\ H_a \cup Y_{n-1} \rightarrow H_a \cup \text{head}(\text{keys}[y_{n-2}]) \cup Y_{n-1} & \text{by augmentation with } Y_{n-1} \\ \text{head}(\text{keys}[y_{n-2}]) \cup H_a \cup Y_{n-1} \rightarrow H_{n-1} \cup H_a & \text{augmentation of } \phi_5 \text{ with } H_a. \end{array}$$

Now we are ready to show that $F \models H_{n-1} \cup Y_n \rightarrow H_a \cup Y_n$ and $F \models H_a \cup Y_n \rightarrow H_{n-1} \cup Y_n$. Since every label l in Y_{n-1} is required for x , and unique for y_{n-1} , by Lemma 6.6, the following FDs are consequences of F :

$$\begin{aligned}\phi_{10} &: \text{head}(\text{keys}[y_n]) \cup \{l\} \rightarrow \text{head}(\text{keys}[y_n]) \\ \phi_{11} &: \text{head}(\text{keys}[y_n]) \rightarrow \text{head}(\text{keys}[y_n]) \cup \{l\}.\end{aligned}$$

Therefore, by the augmentation and transitivity rules with ϕ_1 and ϕ_2 we obtain:

$$\begin{aligned}\phi_{12} &: H_a \cup Y_n \cup Y_{n-1} \rightarrow H_a \cup Y_n \\ \phi_{13} &: H_a \cup Y_n \rightarrow H_a \cup Y_n \cup Y_{n-1}.\end{aligned}$$

Then, we can obtain $F \models H_{n-1} \cup Y_n \rightarrow H_a \cup Y_n$ by transitivity of the following FDs:

$$\begin{array}{ll}H_{n-1} \cup Y_n \rightarrow H_a \cup H_{n-1} \cup Y_n & \text{augmentation of } \phi_4 \text{ with } Y_n \\ H_a \cup H_{n-1} \cup Y_n \rightarrow H_a \cup Y_{n-1} \cup Y_n & \text{augmentation of } \phi_8 \text{ with } Y_n \\ H_a \cup Y_{n-1} \cup Y_n \rightarrow H_a \cup Y_n & \phi_{12}.\end{array}$$

We can obtain $F \models H_a \cup Y_n \rightarrow H_{n-1} \cup Y_n$ similarly using ϕ_{13} , ϕ_9 , and ϕ_3 . Therefore, for all $y_i \in V$, $F \models head(keys[y_i]) \rightarrow K_i$ and $F \models K_i \rightarrow head(keys[y_i])$. \square

Proof of Part 3. We now have to show that F is minimum. Let x be an arbitrary keyed variable. We will consider each of the forms of FDs generated at the x -node, and show that if we insert an FD φ in F then $F \setminus \{\varphi\} \not\models \varphi$.

First, consider an FD of the form $\varphi_1 = K \rightarrow S$, where $K = head(keys[x])$, and either $S = \{l\}$, $l : value(y)$, and $l \in unique[x]$, or $S \in descKeys[x]$, and $S = head(keys[y])$, where y is a descendent of x . By construction, $S \not\subseteq K$, and φ_1 cannot be obtained by reflexivity and augmentation of K or S . Therefore, if φ_1 is redundant, it must be the case that there exists a set of fields Y' such that $F \models \{K \rightarrow Y', Y' \rightarrow S\}$. We will show that such Y' does not exist. Given that $\Sigma \models (P(x_r, x), (P(x, y), \{\}))$, then for any variable x' along the path $P(x, y)$, $\Sigma \models (P(x_r, x)/P(x, x'), (P(x', y), \{\}))$ by the *target-to-context* rule for XML key inference. That is, y is also unique for x' . Therefore, we have to show that there exists no x' that is keyed such that $\Sigma \models (P(x_r, x), (P(x, x'), \{\}))$. For if it holds, $unique[x]$ contains $head(keys[x'])$, and F contains both $K \rightarrow head(keys[x'])$ and $head(keys[x']) \rightarrow S$. Thus, $F \models \varphi_1$ by transitivity. Suppose, by contradiction, that there exists an x' that satisfies the condition above. We will show that if this is the case then $\varphi_1 \notin F$. By construction, if $\Sigma \models (P(x_r, x), (P(x, x'), \{\}))$, and $keys[x] \neq nil$, $uniqueUp[x'] \neq nil$. If $uniqueUp[x] = nil$ then $uniqueUp[x'] = x$ and in the computation of $unique[x]$, it is subtracted by $unique[x']$, and $head(keys[x'])$ is inserted in $descKeys[x]$. Therefore, F contains $K \rightarrow head(keys[x'])$,

but not $K \rightarrow S$, a contradiction. If $\text{uniqueUp}[x] = z$ then $\Sigma \models (P(x_r, z), (P(z, x), \{\}))$. By the *context-to-target* rule for XML key inference, $\Sigma \models (P(x_r, z), (P(z, x)/P(x, x'), \{\}))$, that is, x' is also unique under z . Therefore, $\text{uniqueUp}[x] = \text{uniqueUp}[x']$, and by construction, $\text{unique}[x']$ is also subtracted from $\text{unique}[x]$, and $\text{head}(\text{keys}[x'])$ is inserted in $\text{descKeys}[x]$. Thus, $K \rightarrow S$ is not inserted in F , a contradiction. Given that $\varphi_1 \in F$ implies that there exists no x' such that $\Sigma \models (P(x_r, x), (P(x, x'), \{\}))$, $F \setminus \varphi_1 \not\models \varphi_1$.

Now consider FDs of the form $\varphi_2 = K' \rightarrow (K \setminus K')$ generated by $\text{genFDs}(x)$, where $K = \text{head}(\text{keys}[x])$, and K' is another key in $\text{keys}[x]$. Suppose, by contradiction, that φ_2 is redundant. Let $K = K_1^a \cup L_1$, and $K' = K_2^a \cup L_2$, where $K_1^a = \text{head}(\text{keys}[x_1])$, $K_2^a = \text{head}(\text{keys}[x_2])$, and $(L_1 \cup L_2) \subseteq \text{required}[x]$. Observe that $F \not\models K' \rightarrow K_1^a$ because of the null restriction for FD satisfaction. Since by construction, K and K' are minimum, and there exists no other form of FDs with K' on the left-hand side, if $F \setminus \{\varphi_2\} \models \varphi_2$, it must be the case that $F \models K_2^a \cup L_2 \rightarrow L_1$, and $F \models K_2^a \rightarrow K_1^a$. We will show that the latter is true only when $x_1 = x_2$. Observe that if x_1 is an ancestor of x_2 , the FD violates the null restriction. Therefore, it must be the case that x_1 is a descendent of x_2 . But if $F \models K_2^a \rightarrow K_1^a$, x_1 is unique for x_2 . That is, $\Sigma \models (P(x_r, x_2), (P(x_2, x_1), \{\}))$. In this case, since algorithm computeKeys looks for keys starting from the root, L_1 would be combined with K_2^a to form a key for x , and not with K_1^a . That is, K would not be in $\text{keys}[x]$, a contradiction. Therefore, if $F \models K_2^a \rightarrow K_1^a$, then $x_1 = x_2$, and F contains $K' \rightarrow (K \setminus K') = L_1$. Otherwise, $F \not\models K_2^a \rightarrow K_1^a$, and thus $F \setminus \varphi_2 \not\models \varphi_2$.

Next, we have to show that if F contains $\varphi_3 = K \rightarrow (K \cup S)$ then $F \setminus \{\varphi_3\} \not\models \varphi_3$. By construction, $K = \text{head}(\text{keys}[x])$, and either $S = \{l\}$, and $l \in \text{unique}[w]$, or $S = \text{head}(\text{keys}[w])$, where w is an ancestor of x . Observe that in both cases, $F \not\models K \rightarrow S$ because of the null restriction. Suppose, by contradiction, that $F \setminus \{\varphi_3\} \models \varphi_3$. Then there must exist a set of labels V such that $F \models K \rightarrow V \rightarrow S$, and φ_3 is obtained by augmentation of K . If $F \models V \rightarrow S$, by construction, $V = \text{head}(\text{keys}[w'])$, for some ancestor w' of w . Consider first that $S = \{l\}$. Then, l is in $\text{unique}[w']$ and since $\text{keys}[w'] \neq \text{nil}$, when procedure genFDs is processing x , $\text{uniqueAnc}[x]$ contains l . Therefore, $\varphi_3 \notin F$, a contradiction. Now consider that $S = \text{head}(\text{keys}[w])$. Then, by construction, if $F \models V \rightarrow \text{head}(\text{keys}[w])$ then either $\text{uniqueUp}[w] = w'$, or $\text{uniqueUp}[w] = \text{uniqueUp}[w'] = w''$, for

some ancestor w'' of w' . In both cases, since $uniqueUp[w] \neq nil$, φ_3 would not be in F , a contradiction.

To complete the proof, we have to show that if F contains $\varphi_4 = \{l, l'\} \rightarrow l$, $F \setminus \{\varphi_4\} \not\models \varphi_4$. Let l be a label in $field[x]$, and l' a label in $required[w]$, where $l : value(y)$, $y \leftarrow x/P$ is in $Rule(\mathbf{U})$, and $l' : value(w')$, $w' \leftarrow w/@a$. If $l \notin unique[x]$ then there exists no FD in F , except of this form, with l on the right-hand side. Therefore, $F \setminus \varphi_4 \not\models \varphi_4$. We have to show that this is true also when $l \in unique[x]$. If φ_4 is obtained by augmentation from F then it must be the case that $F \models l' \rightarrow l$. But this is true only if for some variable w , $\{l'\}$ is in $keys[w]$ and l is in $unique[w]$. If this is the case, by construction, both when $w = x$ and when w is an ancestor of x , φ_4 is not included in F . Therefore, if $F \setminus \varphi_4 \models \varphi_4$, it must be obtained by transitivity. That is, there exists a set of labels V such that either $F \models \{l, l'\} \rightarrow V$ or $F \models l' \rightarrow V$, and moreover, $F \models V \cup \{l\} \rightarrow l$. Suppose first that there exists a set of labels V such that $F \models \{l, l'\} \rightarrow V$. But then $\{l, l'\}$ must be in $keys[x]$, and $V = l''$, where $l'' \neq l$. That is, $required[x]$ must contain at least one additional field besides l and thus $required[x] \neq \{l\}$. But if this is the case, φ_4 is not inserted into F , a contradiction.

Suppose now that there exists V such that $F \models l' \rightarrow V$. For the implication to hold, $\{l'\}$ is in $keys[w]$. But if this is the case, $\{l', l\} \rightarrow l$ is only included in F if for all $l'' \in required[w]$, $\{l''\}$ is in $keys[w]$. We will show that if φ_4 is in F there exists no set of labels $V \neq \{l'\}$ such that $F \models l' \rightarrow V$ and $F \models V \cup \{l\} \rightarrow l$. Thus, $F \setminus \varphi_4 \not\models \varphi_4$. Suppose, by contradiction, that V exists. By assumption, for every label $l'' \in required[w]$, $\{l''\}$ is in $keys[w]$. Thus, if for one label $l'' \in required[w]$, $F \models \{l'', l\} \rightarrow \{l\}$ then the implication is true for all required labels in $required[w]$. This is because by Lemma 6.7, for any $\{l''\}$ in $keys[w]$, $F \models l'' \rightarrow head(keys[w])$, and $F \models head(keys[w]) \rightarrow l$. Therefore, by transitivity, for any $l'' \in required[w]$, $F \models \{l'', l\} \rightarrow l$. By construction, if φ_4 is in F then for no other $l'' \in required[w]$, $\{l'', l\} \rightarrow l$ is in F . Therefore, if $F \models l' \rightarrow V$ then V must be a key of some descendent w' of w that is unique under w . But if this is the case, $descKeys[w] \neq \{\}$ and φ_4 would not be included in F . Therefore, there exists no $V \neq \{l'\}$ such that $F \models l' \rightarrow V$ and $F \models V \cup \{l\} \rightarrow l$, and thus, $F \setminus \varphi_4 \not\models \varphi_4$. This completes the proof that F is indeed a minimum cover of all FDs propagated from Σ . \square

6.5 Experimental Study

The various algorithms presented in this chapter have been implemented in a prototype system, which is being used at the Penn Center for Bioinformatics to process gene expression data. We have conducted a number of experiments for the following purposes:

1. to evaluate the performance of Algorithm `propagation` and Algorithm `minimumCover` in practice;
2. to compare the performance of Algorithm `minimumCover` versus `naive` when computing a minimum cover of FDs propagated from a set of XML keys;
3. to compare the cost of Algorithm `propagation` versus a generalization of Algorithm `minimumCover` for checking XML key propagation;
4. to analyze the impact of different data sets (the size of the relational schema \mathbf{U} , the size of table tree/transformation, the number of XML keys) on the performance of Algorithms `propagation` and `minimumCover`.

As will be seen shortly, the results of our experiments show that both Algorithm `propagation` and Algorithm `minimumCover` work well in practice: they take merely a few seconds even given large relational schema, transformation/table tree and XML keys as input. This demonstrates that despite their $O(m^2n^4)$ and $O(m^4n^4)$ worst-case complexities, the algorithms are efficient in practice. For computing minimum cover, Algorithm `minimumCover` is several orders of magnitude faster than Algorithm `naive`, and for checking key propagation Algorithm `propagation` significantly outperforms the generalization of Algorithm `minimumCover`. Our experimental results also reveal that Algorithm `minimumCover` is more sensitive to the number of XML keys than to the size of the transformation. This is a rather pleasant discovery, since in many applications the number of keys does not change frequently whereas a relational schema may define tables with a variety of different arities (number of fields). Our results also show that Algorithm `propagation` has a surprisingly low sensitivity to the size of the transformation, and that its execution time grows linearly with the size of XML keys.

6.5.1 Experimental Testbed

To explore the impact of different types of transformation rules and XML key sets on the performance of our algorithms, we designed a synthetic generator of transformations and XML keys. The generator produces experimental data sets based on the following user-defined parameters: the number of fields in a relation, the depth of a table-tree (i.e., the size of a transformation), and the number of XML keys. The choice of these parameters was guided by the statistics of applications at the Penn Center for Bioinformatics involving importing data in MAGE-ML (a standard designed to represent gene expression data and relevant annotations) into GUS (a pre-existing relational database).

All experiments were conducted on the same 1.6GHz Pentium 4 machine with 512MB memory. The operating system is Linux RedHat v7.1 and the program was implemented in C++ using Qt Library 3.0, running in the KDE desktop environment. The execution times reported in the next section were obtained after running the programs for at least 5 times for each setting.

6.5.2 Experimental Results

We next present the results of three experiments, conducted for different values of the following parameters: the depth of the table-tree (**depth**), the number of keys (**keys**), and the number of fields (**fields**).

The first experiment evaluates the performance of the two algorithms developed in Section 6.4 for computing minimum cover. Figure 6.14 presents the results for two settings: while varying the number of fields in **U**, the first curve depicts the execution time of Algorithm **naive** for **depth** = 5 and **keys** = 5, while the other two curves plot the execution time of Algorithm **minimumCover** for **depth** = 5 and **keys** = 5 as well as for **depth** = 10 and **keys** = 10. We have also conducted the experiments for different values of **depth** and **keys**, and the results are consistent with Figure 6.14. These results tell us the following. First, the average complexity of Algorithm **minimumCover** in practice is much better than its $O(m^4n^4)$ worst-case complexity given in Section 6.4, where m and n are the sizes of

	depth:5 keys:5		depth:10 keys:10
field	naive	minimumCover	minimumCover
5	0.302	0.05	0.2877
6	0.866	0.061	0.2938
7	2.36	0.067	0.3161
8	6.804	0.074	0.3297
9	20.259	0.089	0.3398
10	55.407	0.102	0.368
15		0.189	0.5448
20		0.341	0.834
25		0.449	1.1985
30		0.583	1.6263
50			3.364
100			8.436
200			32.903
500			132.15

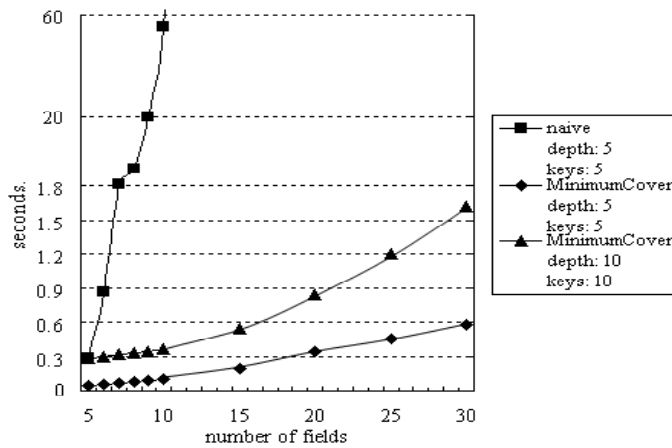


Figure 6.14: Time for computing minimum cover in seconds

the transformation and XML keys respectively. Consider, for example, the execution time of the algorithm for `depth = 10` and `key = 10` given in Figure 6.14. When the number of fields is increased (which corresponds roughly to increasing the size of the transformation), the execution time grows in the power of two in average instead of in the power of four. Second, the table of Figure 6.14 shows that the algorithm needs less than 35 seconds for 200 fields, and a little over 2 minutes even for 500 fields. Since in most applications the number of fields in a relation is much less than 500 – in the GUS database [DCB⁺01] it varies from 7 to 50 with an average of 15 (for which Algorithm `minimumCover` takes less than 1 second) – we can say that Algorithm `minimumCover` performs well in practice.

Third, predictably the performance of Algorithm `minimumCover` is much better than Algorithm `naive`. For example, when the number of fields is incremented by 5, the execution time of `minimumCover` at most doubles, while for `naive` it grows almost two-hundred-fold.

We next consider checking XML key propagation. An algorithm for doing so, Algorithm `propagation`, was presented in Section 6.3. An alternative algorithm can also be developed by means of Algorithm `minimumCover` as follows: Given a transformation σ , a set of keys Σ , and an FD $\phi = Y \rightarrow Z$, the algorithm first invokes `minimumCover`(Σ, σ) to compute a minimum cover F_{mc} of all the FDs propagated; it then checks whether or not F_{mc} implies ϕ using relational FD implication, and whether all the fields in Y are guaranteed to have a non-null value when none of the fields in Z contains `null`. It returns `true` iff these conditions are met. In what follows, we refer to this generalized algorithm as `GminimumCover` since the performance is roughly comparable to the original algorithm.

Our second experiment serves two purposes: to compare the effectiveness of these two algorithms for checking key propagation, and to study the impact of the depth of the table-tree (`depth`) on the performance of Algorithms `propagation` and `GminimumCover`. Figure 6.15 depicts the execution time of these algorithms for `field = 15` and `keys = 10` with `depth` varying from 2 to 15. These parameters were chosen based on the actual average table size of the GUS database [DCB⁺01] and the average tree depth found in real XML data [Cho02]. The results in Figure 6.15 reveal the following. First, Algorithm `propagation` works well in practice: it takes merely 0.05 second even when the table tree is as deep as 15. Second, these algorithms are rather insensitive to the change to `depth`. Third, `propagation` is much faster than `GminimumCover` for checking key propagation, as expected. Although the actual execution times of the algorithms are quite different, the ratios of increase when the depth of the table-tree grows are similar. This is because in both algorithms the depth determines how many times Algorithm `implication` is invoked, and because the complexity of Algorithm `implication` is a function of the size of the XML keys, which grows when the depth of the table tree gets larger.

Our third experiment demonstrates how the number of XML keys (`keys`) influences the

field:15 key:10				
depth	propagation		GminimumCover	
2	0.01613	ratio of increase	0.191	ratio of increase
4	0.0171	6.0%	0.281	47.1%
6	0.01977	15.6%	0.382	35.9%
8	0.02413	22.1%	0.484	26.7%
10	0.0297	23.1%	0.586	21.1%
15	0.05097	71.6%	0.928	58.4%

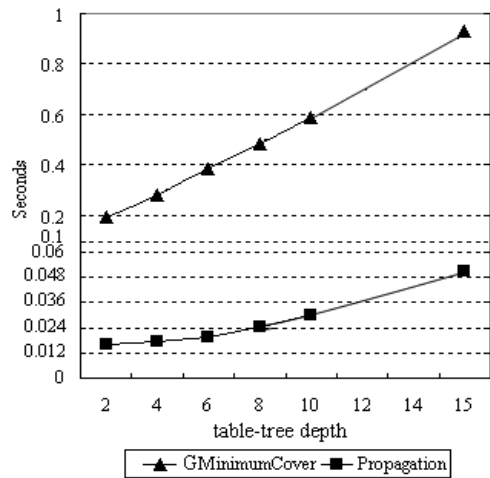


Figure 6.15: Effect of depth of the table tree on the time for computing XML key propagation

performance of Algorithms `propagation` and `GminimumCover` when checking key propagation. Here we choose `fields = 15` and `depth = 10`, again based on the statistics of the GUS database [DCB⁺01] and XML data [Cho02]. The results (Figure 6.16) show that increasing the number of keys has a bigger impact on Algorithm `GminimumCover` than on `propagation`, in which the growth of the execution time is almost linear. In fact, additional experiments tell us that for `depth = 10` and `keys = 50`, Algorithm `GminimumCover` runs in under 2 minutes for 200 fields, but when increasing the number of keys to 100, its execution time is over 4 minutes for relations with 150 fields. In contrast, Algorithm `propagation` runs in both settings in less than 5 seconds. In addition, for 1000 fields, which is the maximum number of fields allowed by Oracle [Ora01], the execution time of `propagation` is 85 seconds on average for 50 keys, and 142 seconds for 100 keys.

field:15 depth:10				
keys	propagation		GminimumCover	
5	0.02148	ratio of increase	0.279	ratio of increase
10	0.0263	22.5%	0.42	50.5%
15	0.03215	22.2%	0.816	94.3%
20	0.04025	25.2%	1.441	76.6%
25	0.04441	10.3%	2.227	54.5%
30	0.05517	24.2%	3.197	43.6%

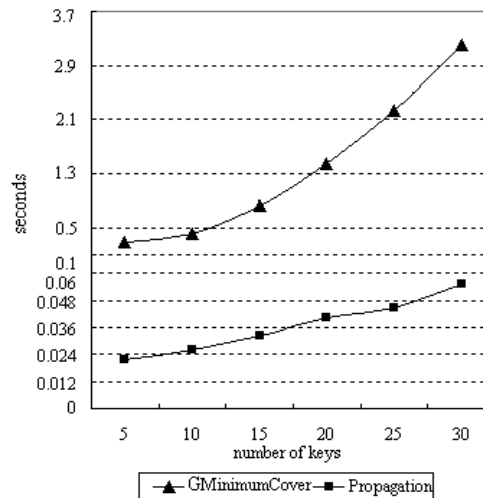


Figure 6.16: Effect of number of keys on the time for computing XML key propagation

A closer look at Algorithm `propagation` reveals that the constant ratio of increase is based on the time needed for executing the calls to Algorithm `implication`. That is, if the depth of the table-tree is fixed, the number of calls is roughly the same for the whole experiment; the increase in running time is based on the the performance of Algorithm `implication`, which depends on the size of the XML keys. The performance of `implication` also has an impact on the Algorithm `GminimumCover`. However, the number of keys has a bigger impact on this algorithm because for each node in the table-tree all the keys are analyzed. Also, by increasing the number of XML keys, the number of FDs in the resulting set is likely to grow, increasing the execution time for minimizing the list of keys and for checking whether an FD is redundant.

In summary, our experimental results show that in practice both Algorithm `minimumCover` and Algorithm `propagation` perform much better than the worst-case complexity analysis

given in the previous sections. In addition, `minimumCover` is much faster than Algorithm `naive` for computing minimum cover of FDs propagated from a set of XML keys, and Algorithm `propagation` should be used for checking XML key propagation instead of Algorithm `GminimumCover`.

6.6 Discussion

To the best of our knowledge, the only other work related to XML constraint propagation is the query rewriting method of MARS [DT03b, DT03a] and the CPI algorithm [LC01]. In [DT03b], a chase/backchase method for query rewriting of [PT99, DPT99] has been extended for a semistructured data model. The method can be used for determining constraint propagation through views when views are expressed in XBind, an XML analog of relational conjunctive queries, and dependencies are XICs (XML Integrity Constraints), an analog of embedded dependencies for XML. However, [DT03b, PT99] do not consider computation of a minimum cover for propagated FDs, and the generality of XICs does not allow efficient analysis when being used to study the propagation of XML keys. CPI is orthogonal to our work because it derives constraints from DTDs (the so-called type constraints), while we propose a transformation language and key propagation analysis that is independent of a DTD. If both DTDs and XML keys are present, we could combine both approaches for determining constraint propagation. This work also parallels that of [AMN⁺01, AMN⁺03], which investigates propagation of type constraints through queries.

In the relational context, the problem of mapping constraints through views has been well-studied [Klu80, KP82, MMS79, BV84b], and polynomial time algorithms for finding a minimum cover for a set of FDs have been developed in [Mai80] and [BB79]. These algorithms take FDs as input and thus the complexity of the algorithms is a function of the size of the FDs. In contrast, in the context of XML key propagation the goal is to compute a cover for a (possibly exponentially large) set of FDs that are mapped from XML keys. These FDs are not provided as input; thus the size of the FDs cannot be used as a parameter of the complexity function. Our problem is related to finding a cover for the FDs embedded in a subset of a relation schema, which is inherently exponential [Got87].

It is worth mentioning that the problem of computing embedded FDs cannot be reduced to ours since the XML key language cannot capture relational FDs, and vice versa.

Several approaches have been explored for using a relational database to store XML documents, and they can be classified into two categories: those that represent the document as a graph [MFK01] or set of paths [SKWW00], and those that capture the structure of the data by grouping together certain subelements [STZ⁺99, BFRS02]. For those in the first category, very little can be said about how constraints in the XML document are propagated to the database, since the connection between subelements is lost when mapped to the relational database. In contrast, approaches in the second category take advantage of DTDs [STZ⁺99] or XML Schema [BFRS02] to generate relational storage mappings that maintain related data in the same relation, and hence dependencies may be preserved. Our framework allows the grouping of subelements by means of variable scoping rules, and does not require a DTD or any other type of schema. Our algorithms for deriving functional dependencies from XML keys could be combined with techniques in the second category to automatically generate mappings through which constraints can be reasoned about.

The transformation language presented in Section 6.1 is similar to that of Stored [DFS99]. While our path language for transformation specifications is more expressive, Stored has features that we do not support, such as label variables. The new release of Oracle (9i) [Ora01] also supports a new datatype called XML Type with member functions to traverse it using XPath. With this functionality, a relation can be created and populated using a Table function in a way similar to our mapping language.

The results presented in this chapter are not only useful for refining the relational design for XML storage, but also for optimizing queries and in understanding XML to XML transformations.

Chapter 7

Conclusion

This chapter summarizes the results reported in this dissertation and identifies further research directions.

7.1 Contributions

The main goal of this dissertation is to investigate how constraints can be used to check the consistency of data being exchanged between different sources. Data exchange involves transformations of data, and therefore the “transformed” data can be seen as a view of its source. Thus, the problem we have investigated is how constraints are propagated to views, when the data involved is not restricted to relational tables, but may be hierarchically structured in several levels of nesting. The ability to determine constraint propagation relies on the ability to determine constraint implication. This is because the validity of a constraint on the view may not result directly from constraints defined on the source data, but from their consequences. Therefore, we have started the dissertation by investigating two forms of constraints: nested functional dependencies (NFDs) and keys for XML, and their implication problems.

Nested Functional Dependencies (NFDs) We have presented a definition of functional dependencies (NFD) for the nested relation model. NFDs naturally extend the

definition of functional dependencies for the relational model by using path expressions instead of attribute names. The meaning of a path expression of the form $A_1 : \dots : A_n$ is that each A_i , $i < n$ is an attribute of set type, and A_{i+1} denotes an element of the set. The meaning of NFDs was given by defining their translation to logic, and it involves both equality of base types as well as equality of set types. NFDs provide a framework for expressing a natural class of dependencies in complex data structures. In particular, in our definition of NFDs, both inter- and intra-set dependencies can be expressed. NFDs can also express that a given set is expected to be a singleton, and that sets should not share elements.

We presented a set of inference rules for NFDs that are sound and complete for the case when no empty sets are present. Although for simplicity we have adopted the nested relational model, and the syntax of NFDs is closely related to this model, allowing nested records or sets would not change the inference rules presented significantly.

Keys for XML A definition of keys for XML has been presented, which allows one to define both global (*absolute*), as well as local (*relative*) keys. That is, keys that uniquely identify nodes in the context of the entire XML tree, and keys that identify nodes in subtrees of the document, respectively. Our definition of keys is independent of any type specification, and it is more expressive than previous proposals of the XML Standard [BPSM98], XML Data [Lay98b], and XML Schema [TBMM01]. Similar to NFDs, keys also involve path expressions, but path expressions here may contain not only element/attribute labels, but also “//”, a combination of wildcard and Kleene closure. That is, it denotes the traversal of an arbitrary number of labels. Two notions of equality were used to define keys: node identity and value equality of complex values, which in this case involves tree structures. In summary, our keys express that a set of key values (that can be tree values) determines the identity of a node in a context, that can either be a subtree or the entire XML tree.

Since a relative key alone does not allow one to uniquely identify a node in the entire XML tree, we have introduced the notion of transitive keys. That is, a set of XML keys that identifies unique contexts up to the root node. A comparison analysis between our keys and the one proposed by XML Schema have been presented, as well as a discussion

on alternative definition of keys. One of the alternatives discussed is one in which keys determine value equality of trees instead of node identity. This is useful when one wants to maintain XML data in “non-second-normal-form”. That is, when it is preferable to allow redundancy of data rather than split it in different parts of the document. Since this form of keys is closer to the definition of NFDs, we have also presented a comparison analysis between these two different notions of constraints.

Another important characteristic for keys is whether key values are required to exist or not. We have identified two notions of keys, both important in their own right: *strong keys*, where key values must exist and be unique, which is similar to the notion of keys in the relational model; and *weak keys*, where key values may not exist, and can define a set of values, which is to cope with the semi-structured nature of XML. We denoted as \mathcal{K} the class of weak keys, and \mathcal{K}_{att} the class of strong keys with key values restricted to be simple attributes. We then presented a number of results for decision problems of the key languages \mathcal{K} and \mathcal{K}_{att} , that we enumerate below:

1. Any finite set of keys in \mathcal{K} or in \mathcal{K}_{att} is finitely satisfiable; that is, given any finite set Σ of keys in these languages, there always exists a finite XML tree that satisfies Σ .
2. There exists a sound and complete set of inference rules for determining containment of expressions in our path language. Moreover, given two path expressions P and Q , deciding whether $P \subseteq Q$ can be determined in $O(|P| |Q|)$ time, where $|P|$ is the length of P .
3. There exists a sound and complete set of inference rules for determining implication of *weak absolute keys*. Given a set of absolute keys $\Sigma \cup \{\varphi\}$, it can be determined whether $\Sigma \models \varphi$ in $O(n^5)$ time, where n is the size of keys involved.
4. There exists a sound and complete set of inference rules for determining implication of keys in \mathcal{K} . Given a set of \mathcal{K} constraints $\Sigma \cup \{\varphi\}$, there is an algorithm that determines whether $\Sigma \models \varphi$ in $O(n^7)$ time.
5. There exists a sound and complete set of inference rules for determining implication of keys in \mathcal{K}_{att} . Given a set of \mathcal{K}_{att} constraints $\Sigma \cup \{\varphi\}$, there is an algorithm that

determines whether $\Sigma \models \varphi$ in $O(n^4)$ time.

The ability to reason about keys efficiently gave us the basis to develop a framework to address the problem that motivated our work. That is, to determine propagation of XML keys to functional dependencies defined on a relational view of XML data.

Propagation of XML constraints to relations Being able to determine whether an FD is valid on relations that store XML data is important in two different scenarios: one is to check the consistency of imported XML data with a predefined relational schema; the other is to design a relational database from scratch or to re-design it to fit the constraints, and thus preserve the semantics, of data being imported. The framework we have presented is based on inferring functional dependencies from XML keys through a given mapping (transformation) of XML data to relations.

Our first contribution consisted of undecidability results that show the difficulty of the generic problem of XML constraint propagation. These negative results gave practical motivation for the restrictions adopted in the framework. In particular, one result showed that it is impossible to effectively propagate all forms of XML constraints supported by XML Schema, which include keys and foreign keys, even when the transformations are trivial. This motivated our restriction of constraints to strong XML keys in \mathcal{K}_{att} . Another undecidability result showed that when the transformation language is too rich, XML constraint propagation is also not feasible, even when only keys are considered. Since XML to relational transformations are subsumed by XML to XML transformations expressible in XML query languages, this negative result applies to most popular XML query languages such as XQuery [Cha01] and XSLT [Cla99].

Our second contribution was a polynomial time algorithm for checking whether an FD on a predefined relational database is propagated from a set of XML keys via a transformation. Given the undecidability results mentioned earlier, we considered a class of XML keys that is a subset of keys in XML Schema [Tho02] and includes those commonly found in practice. We also presented a simple mapping language that is capable of specifying transformations from XML data to relations of any predefined schema and is independent of DTDs and other schema information for XML. In this setting, we developed the first algorithm for

checking XML key propagation. It is based on the sound and complete inference system for keys in \mathcal{K}_{att} .

Our third contribution was a polynomial-time algorithm that, given a universal relation specified by a transformation rule and a set of XML keys, finds a minimum cover for all the functional dependencies mapped from XML keys. This allows us to normalize the design of the relational schema for storing XML data, and optimize query and update operations on XML data stored in relations [KKN04]. Note that the polynomial-time algorithm is rather surprising, since it is known that a related problem in the relational context – finding a minimum cover for functional dependencies *embedded* in a subset of a relation schema – is inherently exponential [Got87].

To verify the effectiveness of our method, we also provided experimental results which show that the algorithms are efficient in practice.

7.2 Further Work

There remain a number of questions related to the work of this dissertation that have not been explored, and below we describe some of them.

In Chapter 3 we have presented a set of inference rules for NFDs that are sound and complete when no empty sets are present. It would be interesting to investigate ways of relaxing this assumption. One possible approach is require the user to define which set-valued paths are known to have at least one element. We believe this is a natural requirement to make, since definition of cardinality has long been recognized as integral part of schema design [Che76] and is part of the DDL syntax for SQL (NON-NULL). Generalizing the inference rules to this case would allow us to reason about constraints for a larger family of instances.

In Chapter 5 we have investigated two key constraint languages introduced in Chapter 4 and studied their associated (finite) satisfiability and (finite) implication problems in the absence of DTDs. For further research, a number of issues deserve investigation. First,

despite their simple syntax, there is an interaction between DTDs and our key constraints. As described in Section 4.3, in the presence of DTDs, keys may not be finitely satisfiable. This shows that in the presence of DTDs, the analysis of key satisfiability and implication can be wildly different. It should be mentioned that keys defined in other proposals for XML, such as those introduced in XML Schema [TBMM01], also interact with DTDs or other type systems for XML. This issue was recently investigated in [AFL02a, FL02] for a class of keys and foreign keys defined in terms of XML attributes.

Second, there are more general definitions of key constraints that should be considered. Among them are the ones that require equal keys to imply value-equality on nodes rather than node identity. This is useful in XML documents in which redundancy is tolerated. It is sometimes useful to put the same information in more than once place in an XML document in order to avoid having to do joins to recover this information.

Third, one might be interested in using different path languages to express keys. The containment and equivalence problems for the full regular language are PSPACE-complete [GJ79], and they are not finitely axiomatizable. Another alternative is to adopt the language of [MS99], which simply adds a single wildcard to the path language. Despite the seemingly trivial addition, containment of expressions in their language is only known to be in PTIME. It would be interesting to develop an algorithm for determining containment of expressions in this language with a complexity comparable to the related result established in this dissertation. For XPath [CD99] expressions, it has been shown [MS04, NS03] that it is rather expensive to determine containment of XPath expressions, which is important in the implication analysis of XML keys. For the (finite) axiomatizability of equivalence of XPath expressions, which is important in studying the (finite) axiomatizability of XML key implication, the analysis is even more intriguing [BFK03]. Thus, not surprisingly, reasoning about keys defined in XML Schema is prohibitively expensive: Even for unary keys, i.e., keys defined in terms of a single subelement, the finite satisfiability problem is NP-hard and the implication problem is coNP-hard [AFL02b]. For the entire class of keys of XML Schema, to the best of our knowledge, both the implication and axiomatizability problems are still open.

Fourth, along the same lines as our XML key language, a language of foreign keys needs to be developed for XML. As shown by [FS03, FL02, AFL02a], the implication and finite implication problems for a class of keys and foreign keys defined in terms of XML attributes are undecidable, in the presence or absence of DTDs. However, under certain practical restrictions, these problems are decidable in PTIME. Whether these decidability results still hold for more complex keys and foreign keys needs further investigation.

In Chapter 6 we have presented a framework for determining the propagation of XML keys to functional dependencies defined on the relational view of XML data. One possible extension of our framework is in our transformation language. It can be extended by allowing predicates so that one can select XML data that satisfy certain conditions; our conjecture is that this can be done without significantly increasing the complexity of the algorithms.

One topic for future work is to study the propagation of alternative definitions of keys for XML, in particular keys that determine value equality instead of node equality. This is because in a data exchange format, the existence of redundant data may be tolerable and sometimes desirable to make it easier to understand and to parse. This is in contrast to stored data, for which the normalization theory has been developed for eliminating data redundancy. Although this form of keys can express redundancy of entire subtrees in the XML document, it cannot express redundancy of single element or attribute values. For that, we would need to define a functional dependency. Thus, it would be interesting to investigate restricted forms of FDs for XML that can be reasoned about efficiently, and study how they can be propagated to relations.

Another topic is to study constraint propagation in the presence of types of XML Schema. However, as indicated in [AFL02a, FL02], the interaction between types and constraints makes the analysis of XML key propagation more intriguing and even undecidable. Therefore further work is needed to identify practical restrictions on types and constraints of XML Schema for efficient XML constraint propagation.

It would also be interesting to investigate the propagation of XML keys to NFDs. If the transformation language were modified to keep the identity of every element in the

XML document that is unnested to populate a field in the flat relation, this operation would result in a set of multi-valued dependencies with an id in the left-hand side and the nested attributes on the right-hand side. Then the nested relation could be obtained by a sequence of nesting operations similar to the reconstruction of a nested relation from its flat representation as described in Chapter 3. Given the correspondences between FDs+MVDs and NFDs presented there, we conjecture that it is possible to derive the set of NFDs propagated from XML keys. But this has to be further investigated.

Bibliography

- [AB86] Serge Abiteboul and Nicole Bidoit. Non first normal form relations: An algebra allowing restructuring. *Journal of Computer and System Sciences (JCSS)*, 33(3):361–390, 1986.
- [ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman, 2000.
- [AFL02a] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. On verifying consistency of XML specifications. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 259–270, Madison, Wisconsin, USA, 2002.
- [AFL02b] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. What’s hard about XML Schema constraints? In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 269–278, September 2002.
- [AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [AL04] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. *ACM Transactions on Database Systems*, 29(1):195–232, March 2004.
- [AMN⁺01] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: Typechecking revisited. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 138–149, 2001.

- [AMN⁺03] Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking XML views of relational databases. *ACM Transactions on Computational Logic*, 4(3):315–354, 2003.
- [App98] Vidur Apparao et al. Document Object Model (DOM) Level 1 Specification. W3C Recommendation, October 1998. <http://www.w3.org/TR/REC-DOM-Level-1/>.
- [BA00] Amos Bairoch and Rolf Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28:45–48, 2000.
- [Bak00] Wendy Baker et al. The EMBL nucleotide sequence database. *Nucleic Acids Research*, 28:19–23, 2000.
- [BB79] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):455–469, 1979.
- [BDF⁺02] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for XML. *Computer Networks*, 39(5):473–487, August 2002.
- [BDF⁺03] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and WangChiew Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.
- [Ben00] Dennis Benson et al. GenBank. *Nucleic Acids Research*, 28:15–18, 2000.
- [BFK03] Michael Benedikt, Wenfei Fan, and Gabriel Kuper. Structural properties of XPath fragments. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 79–95, Siena, Italy, January 2003. To appear in *Theoretical Computer Science*.
- [BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

- [BFRS02] Philip Bohannon, Juliana Freire, Prasan Roy, and Jerome Simeon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 64–75, 2002.
- [BFW98] Peter Buneman, Wenfei Fan, and Scott Weinstein. Path constraints on semistructured and structured data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 129–138, 1998.
- [BLS⁺94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *Sigmod Record*, 23(1):87–96, 1994.
- [BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium (W3C), Feb 1998. <http://www.w3.org/TR/REC-xml>.
- [BV84a] Catriel Beeri and Moshe V. Vardi. Formal systems for tuple and equality generating dependencies. *SIAM Journal of Computing*, 13(1):76–98, 1984.
- [BV84b] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.
- [CD99] James Clark and Steve DeRose. XML Path Language (XPath). W3C Working Draft, November 1999. <http://www.w3.org/TR/xpath>.
- [CDHZ03] Yi Chen, Susan Davidson, Carmem Hara, and Yifeng Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 189–200, 2003.
- [Cha01] Don Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. <http://www.w3.org/TR/xquery>.
- [Che76] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

- [Cho02] Byron Choi. What are real DTDs like. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2002.
- [Cla99] James Clark. XSL Transformations (XSLT). W3C Recommendation, November 1999. <http://www.w3.org/TR/xslt>.
- [DCB⁺01] Susan Davidson, Jonathan Crabtree, Brian Brunk, Jonathan Schug, Val Tannen, Chris Overton, and Chris Stoeckert. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.
- [DFHQ03] Susan Davidson, Wenfei Fan, Carmem Hara, and Jing Qin. Propagating XML constraints to relations. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 543–554, March 2003.
- [DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 431–442, 1999.
- [DHP97] Susan Davidson, Carmem Hara, and Lucian Popa. Querying an object-oriented database using CPL. In *Proceedings of the Brazilian Symposium on Databases (SBBD)*, pages 137–153, 1997.
- [DK97] Susan Davidson and Anthony Kosky. WOL: A language for database transformations and constraints. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 55–65, 1997.
- [DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 459–470, 1999.
- [DT03a] Alin Deutsch and Val Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 201–212, 2003.

- [DT03b] Alin Deutsch and Val Tannen. Reformulation of XML queries and constraints. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 225–241, January 2003. Also available at: <http://db.cis.upenn.edu/Publications>.
- [ea02] Paul T. Spellman et al. Design and implementation of microarray gene expression markup language (mage-ml). *Genome Biology*, 3(9), 2002.
- [End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [ER01] EMBL-EBI (European Bioinformatics Institute) and Rosetta Inpharmatics. Gene Expression RFP Response, August 2001. www.gem1.org/omg.htm.
- [FL02] Wenfei Fan and Leonid Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.
- [FS03] Wenfei Fan and Jérôme Siméon. Integrity constraints for XML. *Journal of Computer and System Sciences (JCSS)*, 66(1):254–291, 2003.
- [FSTG85] Patrick C. Fischer, Lawrence V. Saxton, Stan J. Thomas, and Dirk Van Gucht. Interactions between dependencies and nested relational structures. *Journal of Computer and System Sciences (JCSS)*, 31:343–354, 1985.
- [GJ79] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- [Got87] Georg Gottlob. Computing covers for embedded functional dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 58–69, 1987.
- [HD99] Carmem Hara and Susan Davidson. Reasoning about nested functional dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 91–100, 1999.
- [HU79] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

- [IRS76] Harry B. Hunt III, Daniel J. Rosenkrantz, and Thomas G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *Journal of Computer and System Sciences (JCSS)*, 12:222–268, 1976.
- [JS82] Gerhard Jaeschke and Hans-Jorg Schek. Remarks on the algebra of non first normal form relations. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 124–138, 1982.
- [KKN04] Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence? In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 144–155, 2004.
- [Klu80] Anthony Klug. Calculating constraints on relational expressions. *ACM Transactions on Database Systems*, 5(3):260–290, 1980.
- [KP82] Anthony Klug and Rod Price. Determining view dependencies using tableaux. *ACM Transactions on Database Systems*, 7(3):361–380, September 1982.
- [Lay98a] Andrew Layman et al. XML-Data. W3C Note, January 1998. <http://www.w3.org/TR/1998/NOTE-XML-data>.
- [Lay98b] Andrew Layman et al. XML-Data. W3C Note, January 1998. <http://www.w3.org/TR/1998/NOTE-XML-data>.
- [LC01] Dongwon Lee and Wesley W. Chu. CPI: Constraints-preserving inlining algorithm for mapping XML DTD to relational schema. *Data & Knowledge Engineering*, 39(1):3–25, 2001.
- [LDB97] Zoé Lacroix, Claude Delobel, and Philippe Brèche. Object views and database restructuring. In *Proceedings of International Workshop on Database Programming Languages (DBPL)*, pages 180–201, 1997.

- [LS97] Alon Y. Levy and Dan Suciu. Deciding containment for queries with complex objects. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 20–31, 1997.
- [LVL03] Jixue Liu, Millist W. Vincent, and Chengfei Liu. Local XML functional dependencies. In *Proceedings of ACM International Workshop on Web Information and Data Management*, pages 23–28, 2003.
- [Mai80] David Maier. Minimum covers in relational database model. *Journal of the ACM*, 27(4):664–674, 1980.
- [Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
- [Mak77] Akifumi Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 447–453, 1977.
- [MFK⁺00] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Don Olteanu. Agora: Living with XML and relational. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 623–626, 2000.
- [MFK01] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Pushing XML queries inside relational databases. Tech. Report no. 4112, INRIA, 2001.
- [MMS79] David Maier, Alberto Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979.
- [MNE96] Wai Y. Mok, Yiu-Kai Ng, and David W. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM Transactions on Database Systems*, 21(1):77–106, March 1996.
- [MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 277–295, 1999.

- [MS04] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, January 2004.
- [NS03] Frank Neven and Thomas Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 315–329, 2003.
- [Ora01] Oracle Corporation. *Oracle9i Application Developer’s Guide - XML, Release 1 (9.0.1)*, 2001.
- [OY87] Z.Meral Ozsoyoglu and Li-Yan Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.
- [PDST00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 273–284, 2000.
- [PT99] Lucian Popa and Val Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 39–57, 1999.
- [RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
- [RKS88] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.
- [RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML Query Language (XQL). Workshop on XML Query Languages, December 1998.
- [Sha01] Jayavel Shanmugasundaram et al. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.
- [SKWW00] Albrecht Schmidt, Martin L. Kersten, Menzo Windhouwer, and Florian Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings*

- of the *International Workshop on the Web and Databases (WebDB)*, pages 47–52, 2000.
- [ST04] Lawrence V. Saxton and Xiqun Tang. Tree multivalued dependencies for XML datasets. In *Proceedings of International Conference on Web-Age Information Management (WAIM)*, pages 357–367, 2004.
- [STZ⁺99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 302–314, 1999.
- [TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. W3C Working Draft, May 2001. <http://www.w3.org/TR/xmlschema-1/>.
- [Tho02] Henry Thompson. Personal communication, 2002.
- [TMD92] Jean Thierry-Mieg and Richard Durbin. Syntactic definitions for the ACEDB data base manager. Technical report, MRC Laboratory for Molecular Biology, Cambridge, 1992.
- [TSS97] Zahir Tari, John Stokes, and Stefano Spaccapietra. Object normal forms and dependency constraints for object-oriented schemata. *ACM Transactions on Database Systems*, 22(4):513–569, December 1997.
- [Ull83] Jeffrey D. Ullman. *Principles of Database Systems - Second Edition*. Computer Science Press, 1983.
- [Var81] Moshe Y. Vardi. The decision problem for database dependencies. *Information Processing Letters*, 12(5):251–254, 1981.
- [VL03] Millist W. Vincent and Jixue Liu. Functional dependencies for XML. In *Proceedings of the Asian Pacific Web Conference (APWeb)*, pages 22–34, 2003.

- [VLL04] Millist W. Vincent, Jixue Liu, and Chengfei Liu. Strong functional dependencies and their applicatiopn to normal forms in XML. *ACM Transactions on Database Systems*, 2004. (to appear).
- [Wad00] Philip Wadler. A formal semantics for patterns in xsl. Technical report, Bell Labs, 2000.
- [Wed92] Grant D. Weddel. Reasoning about functional dependencies generalized for semantic data models. *ACM Transactions on Database Systems*, 17(1):32–64, March 1992.