XML QUERY REFORMULATION OVER MIXED AND REDUNDANT STORAGE

Alin Deutsch

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.

2002

Val Tannen
Supervisor of Dissertation

Benjamin C. Pierce
Graduate Group Chair
To Dana, Florence and my parents
ABSTRACT

XML QUERY REFORMULATION OVER MIXED AND REDUNDANT STORAGE

Alin Deutsch
Supervisor: Val Tannen

XML is widely accepted as the standard for data exchange between businesses on the Internet. However, most corporations publish only selected portions of their proprietary business data as XML documents, and even then only virtually, that is by exposing a schema against which queries can be formulated. In order to be answered, such XML queries must be reformulated as queries on the actual proprietary data. Existing XML publishing systems conform to the Global-As-View data integration scenario, in which the correspondence between published (global) and proprietary (local) data is given by expressing the former as a view of the latter. However, an ideal publishing system should enhance query execution by allowing for redundancy in storage which enables multiple reformulations, some potentially cheaper to execute than others. Redundancy requires the complementary, Local-As-View approach to data integration, in which the proprietary data is expressed as a view of the published data. We are led to consider XML publishing systems according to a combined Global-and-Local-As-View approach. Building such a system means facing the following challenges. Existing reformulation algorithms developed for the Global-As-View scenario are said to perform composition-with-views, and they are seemingly unrelated to reformulation algorithms for the Local-As-View scenario, which do rewriting-with-views. Moreover, it turns out that picking the optimal reformulation among the possible candidates requires query minimization. We present MARS, a system implementing a novel reformulation algorithm which achieves the combined effect of rewriting-with-views, composition-with-views and minimization. The algorithm works even when the proprietary storage is a mix of XML documents and relational databases. We prove a completeness theorem which guarantees that under certain conditions, our algorithm will find a minimal reformulation if one exists. Moreover, we study the complexity of the problem and identify conditions when this algorithm achieves best complexity bounds. We report on experiments that show the practicality of the approach.
Acknowledgements

I am greatly indebted to my advisor, Val Tannen, mentor and friend, who turned my stay at Penn into the most exciting and successful experience. Val taught me how to solve a research problem by identifying the underlying basic principles and deriving the solution from them. He also convinced me that the rewards of completing an ambitious, non-compromising research agenda are well worth the time spent in the process. His ideas and deep insights into the theoretical and experimental aspects crucially contributed to this work. My brightest memory of life at Penn is that of the time we spent working together at the whiteboard, and I already miss the ritual of forgetting my cup of tea on his bookshelf after each meeting.

I thank Dan Suciu, who has influenced all important decisions taken along my career path. As a teacher during my undergraduate years at the Polytechnic Institute Bucharest, Dan was a role model for entire generations of students. When I decided to pursue my graduate studies at Penn, I was following his example and advice. As my mentor during a summer internship at AT&T Research Labs in 1998, Dan introduced me to the area of semistructured and XML data, where I eventually found my thesis topic. I have been the fortunate beneficiary of Dan’s advice ever since.

I thank Peter Buneman for encouraging my research interests, and for educating all Penn database group students by means of spontaneous brainstorming sessions and provocative discussions of visionary ideas. Peter was instrumental in creating a Penn database group culture, which all of us are proud to share (and which boosted the business of the White Dog Cafe and New Deck Tavern as a side-effect).

Thanks to Susan Davidson, for teaching me databases both directly (I took all Penn database classes from her), as well as indirectly, through valuable discussions in the Penn
database seminar. Her questions and comments during the seminar meetings helped me look at a problem from a fresh point of view, and taught me how to assess the practical relevance of any proposed solution. I also thank her for her feedback on my work.

Mary Fernandez was my co-mentor during my AT&T internship. Her example set the highest standards on how to run a successful project, from filtering and structuring ideas during the initial brainstorming sessions, all the way to organizing the code of the prototype, managing the software releases, and presenting the results in scholarly publications. I am indebted to Mary for her feedback on this work.

Scott Weinstein introduced me to Mathematical Logic and Finite Model Theory. His intoxicating enthusiasm in class was so inspiring that the interest in these subjects never left me and I ended up applying in my thesis work some of the tools he taught me. Scott’s feedback on my research was invaluable.

My colleague and PhD sibling Lucian Popa was a staunch comrade in both work and play during the years at Penn. This work wouldn’t have been possible without our collaboration and frequent discussions. When our colleague Arnaud Sahuguet joined our research effort during most of 1999, the perfect team was born, working hard and playing hard together. I had never felt such motivating team spirit before.

Finally, I wish to thank my family and close friends for their support. Most importantly, I thank Dana for her courage.
# Contents

Abstract i

Acknowledgements v

1 Introduction 1

1.1 Reformulation in XML Publishing ......................... 1

1.1.1 Approach to query reformulation ...................... 6

1.2 Contributions of this Thesis ............................. 8

1.3 Organization of this Thesis .............................. 12

2 Using the MARS System: a Detailed Example 14

2.1 Configuring and Tuning MARS ........................... 14

2.2 An Example MARS Application ......................... 16

3 Compiling the Queries 25

3.1 Using a Relational Schema to Capture XML ............... 26

3.2 Relational Compilation of the navigation part of XQueries .... 29

4 Relational Query Reformulation: the C&B Algorithm 31

4.1 The C&B Rewriting Algorithm [DPT99] ................ 31

4.2 A New Completeness Result ............................. 33

4.3 Termination of the Chase ............................... 35

4.4 Adding Union to Queries, Disjunction to Constraints ....... 37

4.4.1 Preliminaries: Unions of Conjunctive Queries ....... 37

4.4.2 Chasing with Disjunctive Embedded Dependencies (DEDs) 38
C.3 A Characterization of Containment over Intended Instances . . . . . . . . . 169
C.4 A Sufficient Condition for the Termination of the Chase . . . . . . . . . . . 171
C.5 Proof Sketch for Theorem 6.2.1 (Relative Completeness) . . . . . . . . . . . 173

D More on XPath Containment
(Details for Chapter 9) 174
D.1 Detailed Treatment for Simple XPath . . . . . . . . . . . . . . . . . . . . . 174
   D.1.1 Upper Bounds . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
   D.1.2 Undecidability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
D.2 Detailed Treatment for Extensions of Simple XPath . . . . . . . . . . . . 181
## List of Figures

1.1 The general problem of query reformulation .............................. 1  
1.2 Hiding proprietary data using a public-as-view approach to XML publishing 3  
1.3 The need for both public-as-view and storage-as-view in XML publishing . 4  
1.4 Configuration for example 1.1.1 ........................................... 5  
1.5 MARS architecture .......................................................... 8  
2.1 Simple MARS Configuration .................................................. 15  
2.2 General MARS Configuration ............................................... 15  
2.3 Motivating example configuration .......................................... 16  
2.4 Motivating example after tuning ............................................. 16  
2.5 Music.rdb (proprietary, stored in RDBMS) ................................. 17  
2.6 Example pieces of Music.xml (published, virtual) ......................... 18  
2.7 A fragment of Vienna.xml (proprietary, stored as native XML) ......... 19  
2.8 A query against the published schema ..................................... 20  
2.9 Encodings across models ...................................................... 21  
2.10 Music.rdb encoded as XML .................................................. 21  
2.11 MusicMap ................................................................. 21  
2.12 RVView produces RV.rdb (redundant, stored in RDBMS) ................. 22  
2.13 XView produces XV.xml (redundant, stored native XML) ............... 22  
3.1 MARS modules described in Chapter 3 ..................................... 25  
4.1 MARS module described in Chapter 4 ..................................... 31  
5.1 MARS module described in Chapter 5 ..................................... 43  
6.1 Fragment of chase result ...................................................... 61  
7.1 Computing a match by evaluating JT_P on SDI ............................ 67
7.2 Speedup of new implementation of C&B ........................................ 71
7.3 Time for reformulation without minimization ................................. 75
7.4 Time for reformulation with minimization ...................................... 75
7.5 Benefit of MARS optimization ....................................................... 77
8.1 XML representation of author entities ........................................... 79
8.2 Relational abstraction of author entities ......................................... 79
8.3 The overall approach in schema abstraction ................................... 81
8.4 Benefit of Schema Abstraction ....................................................... 84
12.1 One Aspect of Data Security ....................................................... 137
D.1 XPath expressions in counterexample D.2.1 ................................. 182
Chapter 1

Introduction

This thesis addresses the problem of reformulation of XML queries when the data is stored in mixed (relational and XML) storage.

The problem of query reformulation is a very general one: given two schemas $P$ and $S$ and a correspondence $CR$ between them, and given a query $Q(P)$ formulated in terms of $P$, find a query $X(S)$ formulated in terms of $S$ that is equivalent to $Q$ modulo the correspondence $CR$.

![Figure 1.1: The general problem of query reformulation](image)

1.1 Reformulation in XML Publishing

Reformulation algorithms have many uses in database technology, but our motivation and specific challenges came from XML publishing. XML is widely accepted as the standard for data exchange between businesses on the Internet. However, most corporations publish
only selected portions of their proprietary business data as XML documents, and even then only virtually, that is by exposing a schema (e.g., a DTD). Clients formulate queries against this XML schema (e.g., in XQuery [W3Ce]) and the publishing system must reformulate these into queries on the actual proprietary data in order to answer them. The proprietary data resides in relational databases (RDB) and in native XML document storage (e.g., if acquired through XML exchange).

Therefore, in this thesis we study the following problem:

**Given:**
- the public schema $P$ as XML
- the proprietary storage schema $S$: mixed (RDB+XML)
- the client query $Q$ formulated over $P$ in XQuery
- the schema correspondence $CR$ between $P$ and $S$

**Find:**
- one or more queries $X$ formulated over $S$, such that
  - $X$ is equivalent to $Q$ under $CR$

We still must specify how to model the schema correspondence $CR$. The same problem arises in data integration, where the global—integrated—schema corresponds to our public schema $P$ and the the local—data sources—schema corresponds to our storage schema $S$. Data integration systems use one of two approaches for specifying the schema correspondence [Hal00, Len02]: “Global-As-View” (GAV) and “Local-As-View” (LAV) with the views themselves (sometimes called mappings) expressed in a query language. In our problem’s terminology we could say public-as-view vs. storage-as-view. As we are about to explain, neither of these two approaches used in isolation is flexible enough for our problem.

The public-as-view approach is convenient for hiding portions of the proprietary data: the view definition can simply project/select them away (see Figure 1.2).

This cannot be done in a storage-as-view approach, since the view’s input is in this case the published data, from which the hidden information is missing. On the other hand, we also want to tune the performance of the publishing system by, e.g., caching
query results or redundantly storing some of the native XML data in relational databases in order to exploit the mature relational technology. The resulting redundancies in the stored data can be easily exploited in the storage-as-view approach and will typically lead to multiple reformulations. Some of these are potentially much cheaper to execute than others, so it is desirable to use application-specific cost models in order to pick the cheapest reformulation to execute. However, existing techniques for the public-as-view approach do not handle such redundancies properly (see related work). We conclude that in common XML publishing scenarios we need schema correspondences specified using a combination of both kind of views, each of them a mapping from a portion of the storage schema to a portion of the public schema, or conversely.

The following example illustrates this combined approach.

**Example 1.1.1 (with mixed & redundant storage)** Consider the proprietary relational database `patient` from Figure 1.4, which is partially published as XML using the public-as-view approach, through the mapping/view `PatientMap` that produces `patient.xml`. We emphasize that `patient.xml` is a virtual document; it is the result that would be obtained if `PatientMap` were run. As discussed above, we typically want to hide some information. For example `PatientMap` will *not* publish the patient name. Moreover, the proprietary data in this example is using *mixed storage* because we assume that it also includes a native XML document `catalog.xml` which associates to each drug a price and some notes (on side-effects, generic alternatives, etc. in general the notes vary wildly between drugs and have a highly irregular structure). This part of the proprietary data is
Figure 1.3: The need for both public-as-view and storage-as-view in XML publishing

published in its entirety through the identity mapping $\text{IdMap}$.

Further, in order to speed up the querying of $\text{catalog.xml}$ we decide to store its nicely structured part (drug name and price info) also relationally, in a table $\text{drugPrice}$. Given the irregular structure of the notes, we decided that it would be counterproductive to store these relationally. Such a choice could very well be made by the STORED system [DFS99] for relational storage of XML, where $\text{drugPrice}$ would be specified (in storage-as-view style) as a materialized view of $\text{catalog.xml}$. We shall call the query defining the view $\text{StorageMap}$.

Finally suppose we maintain a cache of previously answered client queries, and one of the current entries holds the result of query $\text{PrevQ}$, which is interested in retrieving from $\text{patient.xml}$ only the association between drugs and the diagnosis they were prescribed for, disregarding their usage. $\text{PrevQ}$ (like all client queries) returns XML, and we shall call the cache entry holding the result $\text{cacheEntry.xml}$. We see that such performance tuning adds redundant data.

We ended up with a design that combines the public-as-view and storage-as-views approaches. Now assume that a client query is posed against the published documents $\text{catalog.xml}$ and $\text{patient.xml}$, for example to find the association between each diagnosis
and the corresponding drugs’ price. We expect that the system should be able to find at least three reformulations of such a query:
- accessing patient and catalog.xml, or
- accessing patient and drugPrice, or
- accessing catalog.xml and cacheEntry.xml.

(With the drugPrice table stored in the same RDBMS as the patient information and with current technology, the second one is likely the best.)

Exploiting redundancy requires minimization. We saw in example 1.1.1 that redundancy enables multiple reformulations. Some of these are potentially cheaper to execute than others and we want to make sure that our approach will find them. In the presence of redundancy some existing approaches (see section 1.2) will produce only queries that are highly wasteful, e.g., the union of the three reformulations in example 1.1.1, thus defeating the purpose of redundant storage. We call a reformulation minimal (see section 4.1) if it performs a minimal number of scans over source data, in the sense that we cannot remove a scan without compromising equivalence to the original client query; equivalence here is over all databases satisfying the schema correspondence. Note that in general a query has infinitely many reformulations just by trivially adding repeating scans. Note also that if a query has any reformulation then it will have a minimal one as well. Therefore, we will restrict our search to minimal reformulations and hope to provide complete algorithms that will find all of them.
**XML integrity constraints (XICs).** Both the public schema and the storage schema may satisfy integrity constraints. Their presence will never reduce but may often *expand* the space of possible reformulations. While much is known about relational constraints, XML constraint formalisms are still “under construction”. We follow here our proposal [DT01a] for a class of **XML Integrity Constraints (XIC)** whose expressive power captures a considerable part of XML Schema [W3Cc, BDF+01] including keys and “keyrefs” and quite a bit beyond.

**Example 1.1.2** Consider the person/employee database below, and assume the data is published as XML. Instead of showing the view, we describe its effect by showing the corresponding data:

\[
P(\text{name}, \text{ssn}): \begin{align*}
P(\text{"John"}, 1) \\
P(\text{"Jane"}, 2)
\end{align*}
\]

\[
E(\text{ssn}, \text{sal}) : \begin{align*}
E(1, 50K) \\
E(2, 51K)
\end{align*}
\]

\[
<\text{payroll}>
\begin{align*}
<\text{empl}><\text{name}>\text{John}</\text{name}><\text{sal}>50K</\text{sal}></\text{empl}>
\end{align*}
\]

\[
<\text{empl}><\text{name}>\text{Jane}</\text{name}><\text{sal}>51K</\text{sal}></\text{empl}>
\]

\[
</\text{payroll}>
\]

Now consider an XML constraint that says that the text value of \text{name} elements is a key for \text{empl} elements. In the XIC syntax we represent it as a first-order-like assertion in which the predicates are denoted in XPath [W3Cb, DT01a]) (details in section 5.4):

\[
\forall e_1 \forall e_2 \forall n \left[ //\text{empl}(e_1) \land //\text{name}/\text{text}()\langle e_1, n \rangle \land //\text{empl}(e_2) \land //\text{name}/\text{text}()\langle e_2, n \rangle \right] \\
\quad \rightarrow e_1 = e_2
\]

(1.1)

In Chapter 2, we give a more complex XML publishing example, detailing the syntax of the schema correspondence, constraints and client query.

**1.1.1 Approach to query reformulation**

Our general strategy is to “compile” the XML reformulation problem into a relational reformulation problem and then use an algorithm that we have proposed earlier together
with Lucian Popa [DPT99]. The different ingredients of our approach are sketched in the following steps 1–6.

**Step 1** Like [CKS+00] we encode the stored relational schemas into XML (pick one of several straightforward encodings). Then, the DB administrator can define mappings RDB→XML or XML→RDB (e.g., PatientMap and StorageMap in figure 1.4) just by writing them in XQuery. The schema correspondence is given by several **XQuery views** (in both directions).

**Step 2** We take **integrity constraints** into consideration provided the constraints on the relational part are expressed as **disjunctive embedded dependencies** (DEDs), see [AHV95, DT01b] and section 4.1, and the constraints on the XML part are expressed as **XICs**, see example 1.1.2, section 5.4 and [DT01a]. (In fact, in view of the encoding in Step 1, we can also use XICs for constraints between the RDB and XML parts.)

**Step 3** Like [CKS+00, MFK01] we follow [FTS00] in splitting **XQuery = navigation part + tagging template**. According to [W3Ce], in a first phase the navigation part searches the input XML tree(s) binding the query variables to nodes or string values. In a second phase that uses the tagging template a new element of the output tree is created for each tuple of bindings produced in the first phase (see example 3.2.1 below). Previous research has addressed the efficient implementation of the second phase [SKS+01, FMS01]. Only the first phase depends on the schema correspondence so we focus on **reformulating the navigation part of XQueries**.

**Step 4** We define a **generic relational schema** called GReX for encoding XML, see section 3.1. Then, the XML encoding of the stored relational schemas that we picked in Step 1 is captured by a set of DEDs relating these schemas to schema GReX, as explained in section 5.1.

**Step 5** We define a syntactic restriction of XQuery, the **behaved** class that is still very powerful, see section 6.2. We give algorithms that translate: (1) the navigation part of a behaved XQuery into a relational union of conjunctive queries over GReX, call it $B$, (2) the behaved XQuery views in the schema correspondence (Step 1) into sets of relational DEDs over GReX, and (3) the (XICs) from Step 2 also into sets of relational DEDs over

---

1Interestingly, in a mixed RDB + XML situation we encode RDB in XML to make view and query specification user-friendly, but then we encode XML in RDB for the automated query processing!
We now have a relational query $B$ (see Step 5) that needs to be reformulated modulo equivalence under the set of all relational constraints coming from Steps 2, 4, and 5. For this we use the C&B algorithm [DPT99]. We prove new theorems that show that our algorithm is indeed complete in that it finds all minimal reformulations (see section 4.1).

Our approach is summarized in Figure 1.5 which happens to also be describing the architecture of the MARS (mixed and redundant storage) system that implements it (more in Chapter 7). In this thesis, we focus on the relational compilation of (i) the navigational part of client XQueries (described by XBind queries as explained in section 3.2) and (ii) the schema correspondence and additional integrity constraints (grey box in Figure 1.5).

1.2 Contributions of this Thesis

The conceptual contribution of this work to the XML publishing research topic is a uniform solution to the problem of finding minimal reformulations of XQueries, under schema correspondences that combine the public-as-view and storage-as-view approaches, where the views are expressed in XQuery, under mixed storage, RDB and XML, and under additional integrity constraints on both the public and storage schemas.

The solution is made possible by a series of technical contributions (1,2,5 below). In
order to better understand these contributions, we conducted a study of the complexity of the problem, helping us to calibrate our results (contributions 3 and 4). Finally, contribution 6 is a first step towards extending our solution to the case when XQueries have bag semantics.

1. Reduction of this XML problem to a similar problem involving only relational queries and relational dependencies

   We give translation algorithms for this reduction (see step 5 in section 1.1.1). We prove a relative completeness theorem for the translation (Theorem 6.2.1) that says in essence that any existing solution of the XML problem can be recovered from some minimal relational reformulation that is a solution of the relational translated problem.

   In devising a translation with such properties we solved technical difficulties posed by the fact that XQueries (a) contain nested, correlated subqueries, (b) create new elements whose node identity is invented, and (c) return deep recursive copies of entire XML subtrees from the input.

   Coupled with an algorithm for minimization under dependencies (see contribution 2 below), this approach unifies the LAV and GAV data integration scenarios (the public-as-view and storage-as-view publishing scenarios) when integrity constraints are allowed on both schemas, by achieving the combined effect of rewriting-with-views, composition-with-views and minimization. This is done by capturing views in both directions with direction-neutral constraints.

2. A new C&B completeness theorem

   The translated problem consist of finding minimal reformulations of unions of conjunctive queries under sets of disjunctive embedded constraints (section 4.1). We solve this problem with the C&B algorithm. This algorithm was introduced in [DPT99] and extended in [DT01b] to also deal with unions and disjunctions. A limited completeness theorem was shown in [DPT99], for the case when the constraints correspond to just storage-to-public views, no views in the reverse direction and no additional constraints on the schemas. In this thesis we prove a much more general C&B completeness theorem, namely for any set of constraints that yield a terminating chase (Theorem 4.2.2). By combining Theorem 6.2.1 with Theorem 4.2.2 we conclude that
our solution to the XML reformulation problem is overall complete.

3. Calibrating our results Our completeness results hold only for the behaved queries (defined in section 6.2), and for bounded XML constraints (in section 4.3). In fact, the method is applicable to larger classes of queries, views and constraints, as long as we can compile them and apply the chase, being understood that we don’t have completeness guarantees anymore. From a practical perspective, we argue that the features that we cover are in our experience the most common ones anyway.

The limitations of the method are not arbitrary. To calibrate our results we first show that checking minimality under dependencies is as hard as deciding query containment (Proposition 4.2.3). Next, we perform a study of the complexity of deciding containment for the navigation part of XML queries in the presence of constraints (chapter 9). The results of this study are summarized below. Proposition 4.2.3 allows us to use the lower bounds from Chapter 9 to show that the restrictions we have imposed are quite essential. Indeed, we conclude that even modest extensions of the class of behaved XQueries will make our algorithm incomplete (unless $NP = \Sigma_2^P$). We also conclude that even modest use of unbounded XML constraints makes the overall problem undecidable.

4. The study of the complexity of containment under integrity constraints Our core formalism allows in the XQueries expressions from a fragment of XPath that we call simple. Since simple XPaths are expressible by the navigation part of behaved XQueries, the obtained lower bounds/undecidability results carry over to the containment of behaved XQueries.

We propose a new class of XML integrity constraints that is naturally associated to this XPath fragment, namely simple XPath integrity constraints (SXIC). SXICs can express many database-style constraints, including key and foreign key constraints specified in the XML Schema standard proposal, as well as many constraints implied by DTDs. We identify a subclass of bounded SXICs under which containment of simple XPath expressions is decidable, but we show that even modest use of unbounded SXICs makes the problem undecidable. In particular, the addition of (unbounded) constraints implied by DTDs leads to undecidability.
We give tight $\Pi_2^p$ bounds for the simple XPath containment problem and tight NP bounds for the disjunction-free subfragment, while even identifying a PTIME sub-case. We also show that decidability of containment under SXICs still holds for non-simple XPath expressions that contain certain additional features (e.g., wildcard) although the complexity jumps to $\Pi_2^p$ even for the disjunction-free subfragment. The decidability of containment of simple XPath expressions in the presence of DTDs only remains open (although we can show that the problem is PSPACE-hard) as well as the problem for full-fledged XPath expressions, even in the absence of integrity constraints.

We extend our results to some (but not all) of the XPath features that depend on document order.

5. Implementation of the MARS system We have implemented a system, called MARS, based on the C&B algorithm, and showed the method to be practical, i.e. feasible and worthwhile. This entailed a good deal of engineering, which is detailed in Chapter 7, together with an experimental evaluation for the MARS system.

6. Understanding the implications of bag semantics on the C&B XQueries have list semantics. In this thesis, we ignore (most aspects of) order, thus solving the problem of reformulation for XQueries with bag semantics (such XQueries can be explicitly written, using the keyword unordered). It turns out that in most cases, the bag semantics coincides with set semantics, or can be reduced to it by making use of the identities of nodes in the XML tree. There is one exception however, namely the concatenation operator, recently introduced into the XQuery working draft. This operator may introduce duplicate occurrences of the same node identity into the list of variable bindings of an XQuery.

While our motivation is the reformulation of XQueries with bag semantics, we start by considering SQL queries. The extension of our results to XQuery is the object of future research.

We extend the C&B method to rewriting SQL queries (with bag and set semantics) using SQL views, in the presence of integrity constraints. We show that the method is usable in realistic optimizers by extending it to bag, set, and bag-set semantics as well as to grouping views and by showing how to integrate it with standard cost-based optimization.
We understand materialized views broadly, including user-defined views, cached queries and physical access structures (such as join indexes, access support relations, and gmaps). Moreover, our internal query representation supports object features hence the method applies to OQL and (extended) SQL:1999 queries. The extended C&B method supports a general class of integrity constraints that are bag- and set-aware, thus being able to find execution plans using views that do not fall in the scope of other methods. In fact, we prove completeness theorems that show that our method will find the best plan in the presence of common and practically important classes of constraints and views, even when bag and set semantics are mixed (as in rewriting a bag query using a combination of set and bag views).

1.3 Organization of this Thesis

Chapter 2 illustrates the use of the MARS system on a detailed application. This example will be used in subsequent chapters to illustrate our techniques.

Chapter 3 shows (part of) the relational framework to which the MARS solution reduces the XML query reformulation problem. In section 3.1, we show how we use the relational data model to host a generic encoding of the XML data model. Using this encoding, we then show in section 3.2 how to compile the navigational part of a client XQuery to a relational query.

Chapter 4 introduces the C&B algorithm, our new completeness result (Theorem 4.2.2) and our extension (section 4.4) allowing union in the queries and disjunction in the constraints. Proposition 4.2.3 helps us identify conditions under which the C&B algorithm achieves optimal complexity bounds, by showing that checking minimality of a query is as hard as deciding query containment. The complexity of the latter problem is studied in detail in Chapter 9. The proofs of Theorem 4.2.2 and Proposition 4.2.3 are in appendix B.3.

Chapter 5 describes the compilation to relational constraints of the schema correspondence and XML integrity constraints.
Chapter 6 puts together the pieces described in Chapters 3, 4, and 5. We show the resulting XML query reformulation algorithm and an example (in section 6.1) detailing the scenario introduced in Chapter 2. Further, we give our Relative Completeness guarantee (Theorem 6.2.1 in section 6.2). More details on the theory of XQuery reformulation are shown in appendix C.

Chapter 7 describes the engineering involved in the implementation of the MARS system, as well as the experimental evaluation showing that the MARS system is practical.

Chapter 8 introduces the XML-specific technique of schema abstraction (section 8.1) which speeds up the chase and backchase stages using information on what conceptual entities are represented by what XML subtrees. The benefit of employing this technique is evaluated experimentally in section 8.2.

Chapter 9 presents the complexity study of deciding containment of XPath expressions under constraints. Since XPath expressions are particular cases of XBind queries, the lower bounds and undecidability results transfer to containment of XBind queries. By Proposition 4.2.3, these lower bounds transfer to our reformulation algorithm. The detailed treatment and some proofs are relegated to appendix D.

Chapter 10 gives our extension of the C&B method to rewriting SQL queries with SQL views under integrity constraints, for an arbitrary mix of bag and set semantics.

Chapter 11 discussed related work and summarizes our conclusions.

Chapter 12 sketches two directions for future research: reformulation of XML integrity constraints (section 12.1) and data security in XML publishing (section 12.2).
2.1 Configuring and Tuning MARS

Configuring the MARS system First we need to specify the published XML schema, but open standards already exist for this task [W3Ca, W3Cc]. Assuming that the schema of the stored/proprietary data is given, next we need to express the relationship between the proprietary data and the published data. Since in general less information is being published, such a relationship can be fully captured only by mappings from the proprietary schema to the published schema. In fact in the motivating example below we have a configuration that cannot be captured at all with mappings in the opposite direction, i.e., from the published schema to the proprietary schema. Given the mixed storage, for configuring a MARS application we may need to specify:

1. Mappings from stored/proprietary RDB to virtual/published XML (as in Figure 2.1)

2. Mappings from stored/proprietary XML to virtual/published XML (also as in Figure 2.1)

3. More generally, mappings that integrate the stored XML and the stored RDB
Figure 2.1: Simple MARS Configuration

Figure 2.2: General MARS Configuration

(as in Figure 2.2). This is more complicated, but sometimes inevitable: consider a mapping that performs a join between XML and RDB data but then hides (projects away) the join attributes.

The motivating example below corresponds to the simpler case in Figure 2.1 but our system deals with the general case in Figure 2.2.

**Tuning the performance in a MARS application** Indexes and materialized views add redundant data in hopes of speeding up queries. Indexing in RDBMS is well-understood. There is active research on XML indexing (e.g., [GW97, CSF+01]), but already it is clear that systems can make use of the XML analog of what was called *access support relations* in OODBMS [KM90]. These can be expressed as materialized RDB views of XML data. For performance tuning with mixed storage we may need to specify (and materialize):

1. Stored XML views of the stored XML or of the virtual XML. Such views might simply be queries previously asked. Caching the results of selected queries is a standard technique [ACPS96].

2. Stored RDB views of the same. This allows us to rewrite some queries that access both XML and RDB data into just RDB queries. Experience has shown that the “set-oriented” processing in RDB engines is generally better than the “graph-navigation” processing associated with XML [STZ+99].

3. Stored XML views of the stored RDB. This comes up e.g. when an XML warehouse is needed for various reasons [FMS01]. If such a view was materialized, a

---

1This begs the question: why not store it all in an RDBMS? But not all XML data is easily stored this way. See our discussion in section 11.1.
MARS system might be able to take advantage of it, depending on the class of queries and on the quality of the native XML storage [CSF+01].

In section 2.2, we give an example that features the first two kinds of materialized views.

### 2.2 An Example MARS Application

![Image of a diagram showing the motivation example configuration](image1)

![Image of a diagram showing the motivating example after tuning](image2)

Consider the proprietary `Music.rdb` relational database (Figure 2.5) that is published as `Music.xml` using a mapping `MusicMap`. Rather than giving a DTD or an XML Schema [W3C] for `Music.xml` we show in Figure 2.6 some pieces of the virtual XML published data that correspond to the example tuples shown in `Music.rdb`. The DTD or XML Schema can be readily figured out. We will show in due course how to express `MusicMap` but we want to point out here that the information in the `id` and `aid` attributes is not published. It is therefore impossible to capture this relationship through a mapping from published to proprietary.

The mixed storage of this example also includes a natively stored XML document `Vienna.xml` (see Figure 2.7). This part of the proprietary data is published in its entirety through the `identity` mapping `idenMap` and we use the same name `Vienna.xml` for the virtual published data. The entire configuration is shown in Figure 2.3.

Consider now querying the published XML. We formulate such a query in Figure 2.8 using the XQuery [W3C] language. Answering this query requires accessing both the RDB

---

2We ask the forgiveness of the reader for basing this example on a funny song by Tom Lehrer, about Alma Schindler, who was married to Gustav Mahler, Walter Gropius, and Franz Werfel.
Songs, their titles, their authors, with names; not all authors have a maiden name.

Conceptual schema:

\[ \text{song}(\text{title}, \text{aid}) \quad \text{author}(\text{id}, \text{last}, \text{first}) \quad \text{maiden}(\text{aid}, \text{name}) \]

Example tuples:

\[
\begin{align*}
\text{song}(\text{title}=\text{"Kennst Du meine Naechte?"}, \text{aid}=1) \\
\text{song}(\text{title}=\text{"Alma"}, \text{aid}=2) \\
\text{author}(\text{id}=1, \text{last}=\text{Mahler-Gropius-Werfel}, \text{first}=\text{Alma}) \\
\text{author}(\text{id}=2, \text{last}=\text{Lehrer}, \text{first}=\text{Tom}) \\
\text{maiden}(\text{aid}=1, \text{name}=\text{Schindler})
\end{align*}
\]

Figure 2.5: Music.rdb (proprietary, stored in RDBMS)

and the XML stored data. This may change if we tune the system by adding redundant materialized views as follows.

Can we do it all in the RDBMS? Although not shown, assume also that the \(<\text{works}>\) subelement of Vienna.xml is very unstructured, containing works described in various formats, annotations, reviews, anecdote, pictures, etc. It may be counterproductive the store this part of the document in an RDBMS (except perhaps as a CLOB –character large object–), but it makes a lot of sense to store the relationship between the nicely structured parts as a relation RV.rdb namely, the person name and the spouse information. Note that RV.rdb is lossy (loses information) not only because it omits the \(<\text{works}>\) elements, but also because it doesn’t distinguish among composers, architects and writers. The relation RV.rdb does not have to be located in the same RDBMS as Music.rdb (see Figure 2.4) but many queries could benefit if both Music.rdb and RV.rdb are made available to the same relational optimizer. We will show below how to express the view RView that produces RV.rdb.

Can we do it all with XML? Imagine that we store an XML view XV.xml that is defined from the Music.xml part of the published schema (corresponding view expression XView is below). Then we might be able to answer some queries by accessing only XV.xml and Vienna.xml. The diagram of this MARS application after configuration and tuning is shown in Figure 2.4.

The query reformulation problem here is the following: given a query \(Q\) on Music.xml + Vienna.xml, reformulate \(Q\) into a query \(Q'\) on Music.rdb + RV.rdb + Vienna.xml + XV.xml such that \(Q\) and \(Q'\) are equivalent given the definitions of MusicMap, idenMap,
The relationships song-author and author–maiden name are captured here just by the nesting and presence/absence of elements.

Figure 2.6: Example pieces of Music.xml (published, virtual)

RView, and XView. If we had only the mappings MusicMap and idenMap to contend with, we could just compose \( Q \) with the mappings (as is done in [FTS00, CKS+00] for the case of just relational storage). If we had only the views RView and XView to deal with, we could use rewriting-with-views as in [MFK01]. Having all four creates a completely new set of challenges. To continue the example we need to see the mappings and the views.

Expressing Views and Queries. What user-level language(s) should be used for configuring and tuning MARS? As query language expressions, a mapping, a view, and a query are the same thing. Since we use XQuery for the queries on the published XML, is XQuery enough? Clearly XML \( \rightarrow \) XML mappings/views can be given in XQuery but for RDB \( \rightarrow \) XML and XML \( \rightarrow \) RDB we have an interesting choice. None of the two data models, relational or XML, is “included” in the other in the same manner in which, say, relations are a particular case of nested relations. However, each can be generically encoded in the other. Such encodings do not lose information and they come together with query translations that preserve them. They can also be easily decoded, which is essential, for example when a view gives RDB data encoded as XML, data that then must be stored in an RDBMS. Given such encodings, the mixed mappings/views can be actually expressed in the standard query languages of XML and RDB, namely XQuery and SQL, see Figure 2.9.
It turns out that encoding XML as RDB and the resulting SQL views are user-unfriendly. Specifically, what can be written as a short XPath expression corresponds in SQL to FROM and WHERE clauses that are just too large to be forced upon a human user [MFK01]. Therefore, in MARS at user level we choose to encode RDB in XML, as [SKS+01] does 3. There are several simple and friendly encodings of relations as semistructured data or XML [ABS99, Biz01] and any of them can be selected by the MARS administrator. We chose one of these encodings and fixed it for this example. We describe it here by example, by encoding some of the Music.rdb tuples, see Figure 2.10. With this encoding, we give MusicMap, RView, and XView in Figures 2.11, 2.12, and 2.13.

Motivating example (continued). Let $Q$ be the query on published XML shown
“Return the names of creative Viennese men who were married to a lied composer”:

\[
\text{for } \quad \text{$p$ in document(Vienna.xml)}//\text{person, $pn$ in $p$/name, $pt$ in $pn$/text()}
\text{where exists $s$ in $p$/spouse satisfies}
\quad \text{exists $sf$ in $s$/fN satisfies exists $sm$ in $s$/mN satisfies}
\quad \text{exists $c$ in document(Music.xml)}//\text{lied/composer satisfies}
\quad \text{exists $cf$ in $c$/first satisfies exists $cm$ in $c$/maiden satisfies}
\quad \text{$cf$/text() = $sf$/text() and $cm$/text() = $sm$/text()}
\text{return <res>$pt</res>}
\]

Figure 2.8: A query against the published schema

in Figure 2.8. Given the schema mappings and materialized views in our example, $Q$ can be reformulated in many ways. Here are some interesting queries, obviously equivalent to $Q$ in this application:

\[
R_1 = \text{for } \quad \text{$p$ in document(Vienna.xml)}//\text{person, $pn$ in $p$/name, $pt$ in $pn$/text()}
\text{where exists $s$ in $p$/spouse satisfies}
\quad \text{exists $sf$ in $s$/fN satisfies exists $sm$ in $s$/mN satisfies}
\quad \text{exists $a$ in document(encode(Music.rdb)) satisfies}
\quad \text{exists $m$ in $a$/maiden satisfies}
\quad \text{$a$/id/text() = $m$/aid/text() and $a$/first/text() = $sf$/text() and}
\quad \text{$m$/name/text() = $sm$/text()}
\text{return <res>$pt</res>}
\]

\[
R_2 = \text{for } \quad \text{$p$ in document(Vienna.xml)}//\text{person, $pn$ in $p$/name, $pt$ in $pn$/text()}
\text{where exists $s$ in $p$/spouse satisfies}
\quad \text{exists $sf$ in $s$/fN satisfies exists $sm$ in $s$/mN satisfies}
\quad \text{exists $c$ in document(XV.xml)/xvxdb/R/BY satisfies}
\quad \text{exists $cm$ in $c$/maiden satisfies}
\quad \text{$cf$/text() = $sf$/text() and $cm$/text() = $sm$/text()}
\text{return <res>$pt</res>}
\]

The query $R_1$ uses only the original mixed storage proprietary data, Music.rdb and Vienna.xml. $R_2$ uses Vienna.xml and the materialized view XV.xml, both natively stored XML.

Assume now that we know that every person-element in Vienna.xml has a unique name-subelement (a constraints very likely to be implied by the DTD of the document). Also
<table>
<thead>
<tr>
<th>Encode</th>
<th>XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDB</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
</tbody>
</table>

Figure 2.9: Encodings across models

```xml
<rdb>
song><title>Naechte</title><aid>1</aid></song>
song><title>Alma</title><aid>2</aid></song>
<author><id>1</id>
  <last>Mahler</last><first>Alma</first></author>
<author><id>2</id>
  <last>Lehrer</last><first>Tom</first></author>
</rdb>
```

Figure 2.10: Music.rdb encoded as XML

```xml
let $music := document(encode(Music.rdb))
<xdb>
  for $song in $music//song
  return
  <lied>
  {$song/title}
  <composer>
  for $author in $music//author
  where $author/id/text() = $song/aid/text()
  return
  {$author/first}
  <last>
  {$author/last/text()}
  for $maiden in $music//maiden
  where $maiden/aid/text() = $author/id/text()
  return
  <maiden>$maiden/name/text()</maiden>
  </last>
  </composer>
  </lied>
</xdb>
```

Figure 2.11: MusicMap

21
The conceptual schema of RV.rdb:  RV(pn, sfn, smn)...

RView below produces RV.rdb encoded in XML (using the encoding we chose for this example, see also Figure 2.10).

```
<rvrdb>
  for $p in document(Vienna.xml)//person,
      $pn in $p/name, $s in $p/spouse,
      $sfn in $s/fN, $smn in $s/mN
  return <RV><pn>$pn/text()</pn>
           <sfn>$sfn/text()</sfn>
           <smn>$smn/text()</smn></RV>
</rvrdb>
```

Figure 2.12: RView produces RV.rdb (redundant, stored in RDBMS)

...pn is person name, sfn, smn are spouse's first and maiden name.

XView below produces XV.xml as an XML document.

```
<xvxdb>
  for $ld in document(Music.xml)//lied,
     $c in $ld/composer,
  return <R>{$ld/title}
          <BY> {$c/first}
              {$c/last}</BY></R>
</vxvd>
```

Figure 2.13: XView produces XV.xml (redundant, stored native XML)
assume that the name of a person is a key, in the sense that there are no distinct person-elements with the same text value in the contents of their name-subelement (something not expressible by a DTD, but rather by an XML schema).

Then \( Q \) would have an additional reformulation that uses the materialized view \( RV.rdb \) together with \( Music.rdb \) and therefore it can be executed as purely relational query:

\[
R3 = \text{for } v \text{ in document(encode(RV.rdb))/rvrdb/RV}, vnt \text{ in } v/pn/text() \text{ where exists } u \text{ in document(encode(Music.rdb)) satisfies exists } a \text{ in } u/author \text{ satisfies exists } m \text{ in } u/maiden \text{ satisfies } \\
\text{exists } a/id/text() = m/aid/text() \text{ and } a/first/text = v/sfn/text() \text{ and } \\
m/name/text() = v/smn/text() \text{ return } <\text{res}>vnt</\text{res}>
\]

It is quite likely that \( R_1 \) executes slower than any of \( R_2, R_3 \). But each of these two can be faster than the other. This depends on whether \( RV.rdb \) is stored in the same RDBMS as \( Music.rdb \), on how much flexibility we have in storing what might a be a “cached query”, namely \( XV.xml \), and also on current and future progress in efficient XML storage techniques [CSF+01]. Using cost information and heuristics, an optimizer will pick one of these queries toward an execution plan.

In order to support MARS applications, the reformulation algorithm must be powerful enough to find \( R_1, R_2, R_3 \) and perhaps other queries, starting just from \( Q \) and the mappings and views defined in this application. In general we have a space of reformulated queries that needs to be explored. The kind of mappings, views and integrity constraints we consider here cannot be handled together by existing approaches [DFS99, FTS00, CKS+00, MFK01]. Different ideas are needed.

**XML integrity constraints** We emphasise here one of the aspects of MARS applications to which existing techniques do not apply, namely the fact that reformulations exploit the known integrity constraints. Here is our proposal for a syntax in which the constraints making possible the reformulation \( R_3 \) can be written down. The syntax combines the logical shape of the classical relational dependencies [AHV95] with atoms defined by the XPath expressions themselves.

\[
(\text{one.Name}) \quad \forall p \forall n_1 \forall n_2 \ [\text{document(Vienna.xml)}//\text{person}](p) \land [./\text{name}](p, n_1) \land [./\text{name}](p, n_2)
\]
\[ \rightarrow n_1 = n_2 \]

\((\text{someName})\)  \(\forall p \ [\text{document(Vienna.xml)}] \rightarrow \text{person}(p) \rightarrow \exists n \ [\text{/name}](p,n)\)

\((\text{keyName})\)  \(\forall p_1 \forall p_2 \forall t \ [\text{document(Vienna.xml)}] \rightarrow \text{person}(p_1) \wedge [\text{/name/text()}](p_1, t) \wedge [\text{document(Vienna.xml)}] \rightarrow \text{person}(p_2) \wedge [\text{/name/text()}](p_2, t) \rightarrow p_1 = p_2\)

\((\text{oneName})\) says that every \text{person}-element in \text{Vienna.xml} can have at most one \text{name}-subelement. Notice the use of the XPaths \text{document(Vienna.xml)}//\text{person}, \text{/name} in the atoms of the constraint, describing the XML element the variables \(p\), respectively \(n_1, n_2\) are bound to. The equality \(n_1 = n_2\) holds if and only if \(n_1, n_2\) have the same node identity (a formal semantics is given in section 9.2). \((\text{someName})\) on the other hand says that every \text{person}-element in the same document has at least one \text{name}-subelement. While this kind of constraints can be implied by a DTD [W3Ca], constraint \((\text{keyName})\) is not: it says that two persons with the same name are identical (as in they have the same node identity). This is reminiscent of a key constraint in relational databases and can be expressed in XML Schema [W3Cc]. See section 9.2 for more examples.
Chapter 3

Compiling the Queries

Figure 3.1: MARS modules described in Chapter 3

We repeat in Figure 3.1 the architecture of the MARS system, and highlight the parts discussed in this chapter. Recall that the MARS solution consists in reducing the reformulation problem to a relational setting. In section 3.1, we show how we use the relational data model to host a generic encoding of the XML data model. Using this encoding, we then show in section 3.2 how to compile the navigational part of a client XQuery to a set of decorrelated XBInd queries, each of whom is in turn compiled to a relational query. In Chapter 5, we show how to compile the schema correspondence to relational constraints, thus completing the relational reduction.
3.1 Using a Relational Schema to Capture XML

We treat mixed XML+relational storage uniformly by reduction to a relational framework. More specifically, following [DT01a], we shall represent XML documents as relational instances \(^1\) over the schema

\[
\text{GReX} = \{ \text{root}, \text{el}, \text{child}, \text{desc}, \text{tag}, \text{attr}, \text{id}, \text{text} \}.
\]

The “intended meaning” of the relations in GReX (Generic Relational Encoding of XML) reflects the fact that XML data is a tagged tree. The unary predicate \(\text{root}\) denotes the root element of the XML document, and the unary relation \(\text{el}\) is the set of all elements. \(\text{child}\) and \(\text{desc}\) are subsets of \(\text{el} \times \text{el}\) and they say that their second component is a child, respectively a descendant of the first component. \(\text{tag} \subseteq \text{el} \times \text{string}\) associates the tag in the second component to the element in the first. \(\text{attr} \subseteq \text{el} \times \text{string} \times \text{string}\) gives the element, attribute name and attribute value in its first, second, respectively third component. \(\text{id} \subseteq \text{string} \times \text{el}\) associates the element in the second component to a string attribute in the first that uniquely identifies it (if DTD-specified ID-type attributes exist, their values can be used for this). \(\text{text} \subseteq \text{el} \times \text{string}\) associates to the element in its first component the string in its second component.

Relational translation of XML tree navigation. Consider an XPath expression \(q\) defined as \(//a\), which returns the set of nodes reachable by navigating to a descendant of the root and from there to a child tagged “a”. Assume also that we materialize the view \(v\) defined as \(//.//a\), i.e. which contains all “a”-children of descendants of descendants of the root. We can translate \(q,v\) as conjunctive queries \(Q,V\) over schema GReX (see appendix A.2 for details):

\[
Q(y) \leftarrow \text{root}(r), \text{desc}(r,x), \text{child}(x,y), \text{tag}(y,\"a\")
\]

\[
V(y) \leftarrow \text{root}(r), \text{desc}(r,u), \text{desc}(u,x), \text{child}(x,y), \text{tag}(y,\"a\")
\]

Clearly, under arbitrary interpretations of the \(\text{desc}\) relation, the two are not equivalent, and \(Q\) cannot be reformulated to use \(V\). But on intended interpretations, the \(\text{desc} \text{ relation}

\(^1\)We emphasize that this does not mean that the XML data is necessarily stored according to the relational schema GReX. Regardless of its physical storage, we reason about XML data using GReX as its virtual relational view.
is transitive and therefore

\[ R(d) \leftarrow V(d) \]

is a reformulation for \( Q \) using \( V \). Any reformulation algorithm must take into account such constraints as transitivity on the intended models of GReX lest it should miss basic reformulations.

**TIX: Constraints inherent in the XML data model.** Some (but not all!) of the intended meaning of signature GReX is captured by the following set TIX (True In XML) of first-order relational constraints: \(^2\)

1. **(base)** \( \forall x, y \ [ \text{child}(x, y) \rightarrow \text{desc}(x, y) ] \)
2. **(trans)** \( \forall x, y, z \ [ \text{desc}(x, y) \land \text{desc}(y, z) \rightarrow \text{desc}(x, z) ] \)
3. **(refl)** \( \forall x \ [ \text{el}(x) \rightarrow \text{desc}(x, x) ] \)
4. **(el\(_c\))** \( \forall x, y \ [ \text{child}(x, y) \rightarrow \text{el}(x) \land \text{el}(y) ] \)
5. **(el\(_d\))** \( \forall x, y \ [ \text{desc}(x, y) \rightarrow \text{el}(x) \land \text{el}(y) ] \)
6. **(el\(_id\))** \( \forall s, x \ [ \text{id}(s, x) \rightarrow \text{el}(x) ] \)
7. **(el\(_r\))** \( \forall x \ [ \text{root}(x) \rightarrow \text{el}(x) ] \)
8. **(someTag)** \( \forall x \ [ \text{el}(x) \rightarrow \exists t \ \text{tag}(x, t) ] \)
9. **(oneTag)** \( \forall x, t_1, t_2 \ [ \text{tag}(x, t_1) \land \text{tag}(x, t_2) \rightarrow t_1 = t_2 ] \)
10. **(keyId)** \( \forall s, e_1, e_2 \ [ \text{id}(s, e_1) \land \text{id}(s, e_2) \rightarrow e_1 = e_2 ] \)
11. **(oneAttr)** \( \forall x, n, v_1, v_2 \ [ \text{attr}(x, n, v_1) \land \text{attr}(x, n, v_2) \rightarrow v_1 = v_2 ] \)
12. **(noLoop)** \( \forall x, y \ [ \text{desc}(x, y) \land \text{desc}(y, x) \rightarrow x = y ] \)
13. **(oneParent)** \( \forall x, y, z \ [ \text{child}(x, z) \land \text{child}(y, z) \rightarrow x = y ] \)
14. **(oneRoot)** \( \forall x, y \ [ \text{root}(x) \land \text{root}(y) \rightarrow x = y ] \)
15. **(topRoot)** \( \forall x, y \ [ \text{desc}(x, y) \land \text{root}(y) \rightarrow \text{root}(x) ] \)
16. **(inLine)** \( \forall x, y, u \ [ \text{desc}(x, u) \land \text{desc}(y, u) \rightarrow x = y \lor \text{desc}(x, y) \lor \text{desc}(y, x) ] \)
17. **(choice)** \( \forall x, y, z \ [ \text{child}(x, y) \land \text{desc}(x, z) \land \text{desc}(z, y) \rightarrow x = z \lor y = z ] \)

\(^2\)A collection \( D_1, \ldots, D_n \) of XML documents is represented by the disjoint union of schemas GReX\(_i\) and the union of constraints in each TIX\(_i\), where each GReX\(_i\) (TIX\(_i\)) is obtained from GReX (resp. TIX) by sub scripting all relational symbols with \( i \).
The family of \( e_1 \times \_X \) constraints can be understood as “typing” constraints, saying that
\( \text{child,desc} \subseteq e_1 \times e_1 \), that only elements may be pointed to using an \( id \) atom, and that
\( \text{root} \) denotes an element, which by \( \text{(oneRoot)} \) is unique, and by \( \text{(topRoot)} \) has no ancestors
beside itself. \( \text{(someTag)} \) and \( \text{(oneTag)} \) say that every element has precisely one tag. The
treeness of the data model is (partially) enforced by such constraints as \( \text{(oneParent)} \) (every
element has at most one parent), \( \text{(noLoop)} \) (only trivial cycles are allowed), \( \text{(inLine)} \) (all
ancestors of an element are located on the same root-leaf path in the tree), \( \text{(choice)} \) (there
is no intermediate element node on a parent-child path).

Note that these axioms are First-Order incomplete; they don’t even prove
\[
\forall x \forall y \ \text{desc}(x,y) \rightarrow x = y \lor \exists z \ \text{child}(x,z) \land \text{desc}(z,y).
\]

Still they are special because they are sufficient to give an optimal, chase-based decision
procedure for containment of XQueries from the fragment with NP-complete containment
(section 9.2).

Notice that except for \( \text{(inLine)} \) and \( \text{(choice)} \), all constraints in TIX are embedded depend-
dencies (as [AHV95] calls them, but also known as tuple- and equality-generating depen-
dencies [BV84a]) for which a deep and rich theory has been developed. \( \text{(inLine)} \) contains
disjunction but so do XQueries. Extending the theory to disjunctive embedded dependen-
cies is fairly straightforward (see section 4.4.1).

Transitive Closure and Treeness. Observe that \( \text{(base)}, \text{(trans)}, \text{(refl)} \) above only
guarantee that \( \text{desc} \) contains its intended interpretation, namely the reflexive, transitive
closure of the \( \text{child} \) relation. There are many models satisfying these constraints, in which
\( \text{desc} \) is interpreted as a proper superset of its intended interpretation, and it is well-known
that we have no way of ruling them out using first-order constraints, because transitive
closure is not first-order definable. Similarly, the “treeness” property of the \( \text{child} \) relation
cannot be captured in first-order logic. The fact that we can nevertheless decide equivalence
of behaved XQueries (containing descendant navigation) over the intended interpretation
using the constraints in TIX and classical relational (hence first-order) techniques comes
therefore as a pleasant surprise.
3.2 Relational Compilation of the navigation part of XQueries

Example 3.2.1 Consider a document containing book elements, each of whom contain a title and some author subelements. The query below restructures the data by grouping the book titles with each author. The groups appear as item elements, whose writer subelement contains the author name and whose (possibly multiple) title subelements contain all titles (co-) authored by this writer.

Q: 

```xml
<result>
  for $a in distinct(/*/author/text())
  return
  <item>
    <writer>$a</writer>
    {for $b in /*/book
      $a1 in $b/author/text()
      $t in $b/title
      where $a = $a1
      return $t}
  </item>
</result>
```

Q’s computation can be described in two stages. First, all bindings for the variable $a to distinct text values of author elements are computed. Next, a unique result root element is created and for every binding of $a, a new item subelement of this result element is created. Each item element has a writer subelement containing the text $a was bound to, and as many title subelements as are returned by the nested subquery shown in braces. Notice that this query is correlated with the outer query through the occurrence of variable $a. The nested query scans all book elements, returning copies of the title subelements of books whose author $a1 coincides with $a.

Describing the navigational part: XBind queries In order to describe the navigational part of an XQuery, we introduce a simplified syntax that disregards the element construction, focussing only on binding variables and returning them. We call the queries in this syntax XBind queries. Their general form is akin to conjunctive queries. Their head returns a tuple of variables, and the body atoms can be purely relational or are predicates.
defined by XPath expressions with restrictions (see appendix A.1 for their syntax). The predicates can be binary, of the form \([p](x,y)\), being satisfied whenever \(y\) belongs to the set of nodes reachable from node \(x\) along the path \(p\). Alternatively, predicates are unary, of form \([p](y)\), whenever \(p\) is an absolute path starting from the root.

**Example 3.2.2** A naive way of evaluating \(Q\) in example 3.2.1 is via nested loops: for every binding of \$a\, evaluate the nested for loop computing the bindings of \$b,\$a1,\$t\ and output the corresponding element. However, research in evaluating correlated SQL queries suggests an alternative strategy that consists in breaking the query into two decorrelated queries which can be efficiently evaluated separately and then putting together their results using an outer join [SPL96]. We will borrow this technique, obtaining for \(Q\) the two decorrelated XBind queries below. \(Xb_o\) computes the bindings for the variables introduced in the outer for loop (\$a\), while \(Xb_i\) computes the bindings of the variables from the inner loop (\$b,\$a1,\$t\) that agree with some value for \$a\ as computed by \(Xb_o\). Notice that this value of \$a\ is output by \(Xb_i\) as well, in order to preserve the correlation between variable bindings. In the query definitions below, we drop the \$ signs from the variable names.

\[
Xb_o(a) \leftarrow [[[/author/text()]](a)
\]

\[
Xb_i(a,b,a1,t) \leftarrow Xb_o(a), [[[/book](b), [[[/author/text()]](b,a1), [[[/title](t), a = a1 \bullet
\]

**Translating XBind queries to relational conjunctive queries** We have seen so far how to describe the navigational part of an XQuery by a set of decorrelated XBind queries. Each of these XBind queries can be translated to a relational conjunctive query over the schema GReX (recall the example on page 26 and see appendix A.2 for the algorithm). \(Xb_o, Xb_i\) above translate to the following queries:

\[
B_o(a) \leftarrow root(r).desc(r,d).child(d,c).tag(c,author).text(c,a)
\]

\[
B_i(a,b,a1,t) \leftarrow B_o(a), root(r).desc(r,d).child(d,b).tag(b,book),
child(b,c).tag(c,author).text(c,a1), child(b,t).tag(t,title), a = a1
\]
Chapter 4

Relational Query Reformulation: the C&B Algorithm

This chapter describes the core of the MARS system, which is the C&B module (Figure 4.1).

4.1 The C&B Rewriting Algorithm [DPT99]

Capturing views with dependencies. The key observation that enables the uniform
treatment of views and integrity constraints by the C&B algorithm is the fact that conjunctive query views can be captured by embedded dependencies relating the input of the defining query with its output. For example, consider the view defined by

\[ V(x, z) \leftarrow A(x, y), B(y, z) \]

In any instance \( I \) over the schema \( \{A, B, V\} \), the extent of relation \( V \) coincides with the result of this query if and only if the following dependencies are satisfied:

\[
\begin{align*}
(c_V) & \quad \forall x \forall y \forall z \ [A(x, y) \land B(y, z) \rightarrow V(x, z)] \\
(b_V) & \quad \forall x \forall z \ [V(x, z) \rightarrow \exists y \ A(x, y) \land B(y, z)]
\end{align*}
\]

Note that the dependencies are obtained from the view definition, and that they state the inclusion of the result of the defining query in the relation \( V (c_V) \), respectively the opposite inclusion \( (b_V) \).

**Phase 1: the chase. Universal Plan.** Assume that in addition, the following semantic constraint is known to hold on the database (it is an inclusion dependency):

\[
(ind) \quad \forall x \forall y \ [A(x, y) \rightarrow \exists z B(y, z)]
\]

Suppose that we want to reformulate the query

\[ Q(x) \leftarrow A(x, y) \]

In the first phase, the query is chased with all available dependencies, until no more chase steps apply (see section 4.4.2 for a detailed definition of the chase). The resulting query is called the *universal plan*. In our example, a chase step with \( (ind) \) applies, yielding \( Q_1 \) below, which in turn chases with \( (c_V) \) to the universal plan \( Q_2 \):

\[
\begin{align*}
Q_1(x) & \leftarrow A(x, y), B(y, z) \\
Q_2(x) & \leftarrow A(x, y), B(y, z), V(x, z)
\end{align*}
\]

Notice how the chase step with \( (c_V) \) brings the view into the chase result, and how this was only possible after the chase with the semantic constraint \( (ind) \).
Phase 2: the backchase. Subqueries. In this phase, the subqueries of the universal plan are inspected and checked for equivalence with \( Q \). Subqueries are obtained by retaining only a subset of the atoms in the body of the universal plan, using the same variables in the head (see section B.1 for a formal definition).

For example, \( S(x) \leftarrow V(x,z) \) is a subquery of \( Q_2 \) which turns out to be equivalent to \( Q \) under the available constraints, as can be checked by chasing \( S \) “back” to \( Q_2 \) using a chase step with \((b_V)\).

### 4.2 A New Completeness Result

It is not accidental that we discovered a reformulation among the subqueries of the universal plan; in fact, in theorem 4.2.2 we give a theoretical guarantee that all minimal reformulations can be found this way.

**Definition 4.2.1 (Minimal query and reformulation under constraints)** A query \( R \) is minimal under a set of constraints \( C \) (or \( C \)-minimal) if no relational atoms can be removed from \( R \)’s body, even after adding arbitrarily many equality atoms, without compromising the equivalence to \( R \) under \( C \). A query \( R \) is a minimal reformulation of query \( Q \) under \( C \) if it is \( C \)-minimal and equivalent to \( Q \) under \( C \) (\( C \)-equivalent to \( Q \)).

Recalling the above example, \( T(x) \leftarrow A(x,y), V(x,z) \) is not minimal under the constraints \( \{(e_V), (b_V), (ind)\} \), because we can remove the \( A \)-atom (without adding equalities) to obtain \( M(x) \leftarrow V(x,z) \), which is equivalent to \( T \), as can be checked by chasing.

Note that in general a query has infinitely many reformulations just by trivially adding repeated atoms: \( M'(x) \leftarrow V(x,z), V(y,u), x = y, z = u \) is a non-minimal reformulation obtained from \( M \). Note also that if a query has any reformulation then it will have a minimal one as well. Therefore, we will restrict our search to minimal reformulations and hope to provide complete algorithms that will find all of them.

If the cost model used to select the cheapest reformulation is monotonic (i.e. the cost of a query is no less than that of any of its subqueries), the cheapest reformulation is guaranteed to be minimal and we do not give up optimality by restricting our attention to such reformulations. The merit of the algorithm is that, under certain conditions, it
is guaranteed to find all minimal reformulations (see theorem 4.2.2 below) and hence the optimal rewriting under a monotonic cost model.

**Theorem 4.2.2** Let $Q$ be a conjunctive query and $D$ be a set of embedded dependencies. Assume that the chase of $Q$ with $D$ terminates, yielding the universal plan $U$. Then any minimal reformulation of $Q$ under $D$ is isomorphic to a subquery of $U$.

The proof is given in appendix B.1. This result adds significant value to the one in [DPT99], where we showed the completeness of the C&B when only views are allowed (i.e. we allow the constraints capturing the views such as $(c_V), (b_V)$, but no additional integrity constraints such as $(ind)$). See section 4.3 for results guaranteeing the termination of the chase.

**Query reformulation via C&B.** When the C&B is used in the following particular scenario: (i) $Q$ is posed against the public schema $P$, (ii) $D$ gives the correspondence between $P$ and storage schema $S$, and (iii) in the backchase phase we consider only subqueries expressed solely in terms of $S$, we obtain a complete algorithm for finding minimal reformulations.

**Minimization under dependencies.** By theorem 4.2.2, whenever the chase terminates, the C&B algorithm is a complete procedure for minimization of conjunctive queries under dependencies generalizing existing procedures (see related work).

**Calibrating the result.** The C&B’s completeness relies heavily on the decidability of query equivalence under dependencies; during the backchase phase, the C&B performs checks for the equivalence under dependencies of a subquery with the universal plan. But there are known classes of dependencies under which query equivalence is undecidable (a mix of functional and inclusion dependencies [BV81] for instance). A natural question to ask is whether there are complete algorithms even when equivalence is undecidable, by not relying on equivalence checks. The answer is no, as shown by the following:

**Proposition 4.2.3** The problem of deciding minimality of a conjunctive query over all models that belong to some class $C$ and satisfy a set of dependencies is at least as hard as deciding containment of conjunctive queries over the class $C$. 34
The proof is given in appendix B.2. In particular, the class $C$ may be specified as all models satisfying a set of dependencies. Undecidability of containment under dependencies therefore implies that the set of minimal reformulations of a query under dependencies is not recursive.

It turns out that the C&B algorithm is asymptotically optimal even when used as an alternative to classical algorithms for rewriting with views in the absence of additional integrity constraints (such as Minicon): the associated decision problem is checking the existence of a rewriting using solely the views, in the absence of constraints. The C&B-based solution would consist in picking from the universal plan $U$ the maximal subquery that mentions only views, and checking its equivalence to $U$. The complexity analysis reveals that the resulting algorithm is in NP in the size of the query, which is optimal according to [LMSS95].

**Extension: DEDs.** The theorem holds even when $Q$ is a union of conjunctive queries and $D$ is a set of disjunctive embedded dependencies (DEDs) (see section 4.4). Their general form is

$$\forall x [\phi(x) \rightarrow \bigvee_{i=1}^{l} \exists z_i \psi_i(x, z_i)]$$

(4.1)

where $x, z_i$ are tuples of variables and $\phi, \psi_i$ are conjunctions of relational atoms of the form $R(w_1, \ldots, w_l)$ and (in)equality atoms of the form $(w \neq w')$ $w = w'$, where $w_1, \ldots, w_l, w, w'$ are variables or constants. $\phi$ may be the empty conjunction. We call such dependencies disjunctive embedded dependencies (DEDs), because they contain the classical embedded dependencies [AHV95] when $l = 1$. A proper DED is (choice) from TIX. Section 4.4.2 shows how we extend the chase to DEDs, and section 4.4.3 shows the resulting extension of the C&B algorithm.

### 4.3 Termination of the Chase

In general, the chase with arbitrary embedded dependencies may not terminate [AHV95]. However, in all scenarios we have encountered in practice, the chase is guaranteed to terminate because the constraints involved satisfy the following property.
Set of constraints with stratified-witness. Given a set $C$ of constraints, define its chase flow graph $G = (V, E)$, as a directed, edge-labeled graph whose labels can be either $\forall$ or $\exists$. $G$ is constructed as follows: for every relation $R$ of arity $a$ mentioned in $C$, $V$ contains a node $R^i$ ($1 \leq i \leq a$). For every pair of relations $R, R'$ of arities $a, a'$ and every constraint $\forall \bar{x} \left[ \ldots \land R(u_1, \ldots, u_a) \land \ldots \rightarrow R'(v_1, \ldots, v_{a'}) \ldots \right]$ in $C$, $E$ contains the edges $\left( R_i, R'_j \right)_{1 \leq i \leq a, 1 \leq j \leq a'}$. Also, whenever the equality $x = y$ appears in the conclusion of the implication, and $x, y$ appear as the $i, j$-th component of $R$, resp. $R'$, $E$ contains the edge $\left( R_i, R'_j \right)$. Moreover, if for some $j$ the variable $v_j$ is existentially quantified, the edges $\left( R_i, R'_j \right)_{1 \leq i \leq a}$ are labeled with $\exists$, otherwise they are labeled with $\forall$.

We say that a set of constraints has stratified-witness if none of the cycles in its chase flow graph contains an $\exists$-labeled edge. Denoting with $|Q|$ the size of query $Q$, with $a$ the maximum arity of a relation in the schema and with $l$ the maximum number of $\exists$-edges on a path in the chase flow graph, we have the following

**Proposition 4.3.1 (with Lucian Popa)** The chase of any query $Q$ with any set of constraints with stratified-witness terminates, and the size of the resulting query is in $O(|Q|^a l^{a+1})$.

This condition explains many particular cases which have been proven to lead to terminating chases: only functional dependencies (key constraints), total/full dependencies, typed 1-non-total dependencies, typed dependencies with identical sets of total attributes [BV84a]. For instance, if all dependencies are full (have no existential quantifiers), there are no $\exists$ edges in the chase flow graph and the chase terminates. Moreover, $l = 0$ and the chase result size is exponential in $a$.

Remarks.

1. Notice that any pair of inclusion dependencies used to capture a view (recall $(c_V), (b_V)$ from page 32) violates the stratified-witness condition. However, the chase is guaranteed to terminate nevertheless, using the additional key observation that the introduction of the view symbol $V$ by a chase step with $(c_V)$ can never trigger a chase step with $(b_V)$. This effectively breaks the $\exists$-cycle appearing in the chase flow graph.

2. The chase flow graph is similar to the graph used to determine the existence of stratified normal forms for programs in the ILOG language [HY90]. Such programs invent
object identities, just like the chase invents new variables.

4.4 Adding Union to Queries, Disjunction to Constraints

In this section we extend the C&B algorithm to unions of conjunctive queries and dependencies with disjunction.

4.4.1 Preliminaries: Unions of Conjunctive Queries

Conjunctive queries. General form. Conjunctive queries have general form

\[ Q(y_1, \ldots, y_n) \leftarrow \text{body}(x_1, \ldots, x_m) \]

where the right hand side is called the body of \( Q \) and \((y_1, \ldots, y_n)\) is called the head tuple. \( \text{body}(x_1, \ldots, x_m) \), is conjunction of relational atoms of the form \( R(w_1, \ldots, w_l) \) and equality atoms of the form \( w = w' \), where \( w_1, \ldots, w_l, w, w' \) are variables or constants. \( x_1, \ldots, x_m \) denote the free variables of this formula, and they contain \( y_1, \ldots, y_n \).

Semantics. Given a database instance \( I \), a valuation \( v \) from \( Q \) to \( I \) is a function that maps the variables of \( Q \) to constants in \( I \) and is the identity on the constants in \( Q \). Moreover, the image of every relational atom \( R \) of \( Q \) under \( v \) is an \( R \)-tuple in \( I \), and for every equality atom \( u = v \) of \( Q \), \( u, v \) have the same image under \( v \).

The answer of a conjunctive query \( Q \) on an instance \( I \), denoted \( Q(I) \), is the set of all tuples \( t \) for which there is a valuation \( v \) from \( Q \) into \( I \), such that the image of the head tuple under \( v \) is \( t \).

Containment and Equivalence. We say that query \( Q_1 \) is contained in query \( Q_2 \), denoted \( Q_1 \subseteq Q_2 \), if for every instance \( I \), the set \( Q(I) \) is included in the set \( Q(I) \). Two queries are equivalent if they are contained in each other.

Union of Conjunctive Queries. A union of conjunctive queries has general form

\[ Q(y_1, \ldots, y_n) \leftarrow \bigcup_i b_i(x_1, \ldots, x_m) \]

where each \( b_i \) is the body of a conjunctive query, and the same restrictions as for conjunctive queries apply on the head and body variables. The answer of the union on an instance \( I \) is the union of the answers of each conjunctive query \( Q(y_1, \ldots, y_n) \leftarrow b_i(x_1, \ldots, x_m) \) on
The definition of containment of conjunctive queries applies unchanged to unions of conjunctive queries.

4.4.2 Chasing with Disjunctive Embedded Dependencies (DEDs)

Observe that all constraints in TIX are DEDs. In chapter 5 we show how various default encodings of relations as XML as well as views are compiled to DEDs. Similarly, we show there how XML integrity constraints compile to DEDs over the schema of TIX. We give here an example for the latter:

\[ \forall x \left[ \text{desc}(@ gender, x) \land \text{child}(x) \land \text{tag}(x) \land \text{attr}(x, \text{gender}, x) \land x = "m" \lor x = "f" \right] \]

For this reason, in the rest of this thesis, we will use the terms “DED” and “constraints” interchangeably.

Satisfaction of DEDs

The satisfaction of a DED \( d \) by an instance \( I \), denoted \( I \models d \), is defined according to satisfaction of First-Order sentences.

Containment under DEDs and Implication of DEDs

It is known that the problems of implication of embedded relational dependencies [AHV95] and containment of relational conjunctive queries in their presence are inter-reducible [YP82]. The extension of this result to unions of conjunctive queries and disjunctive embedded dependencies (which is what we call constraints) is straightforward:

Containment under DEDs. We say that conjunctive query \( Q_1 \) is contained in conjunctive query \( Q_2 \) under the set of DEDs \( D \) if the answer of \( Q_1 \) is contained in that of \( Q_2 \) on all instances \( I \) that satisfy all the DEDs in \( D \). We denote this property with \( Q_1 \subseteq_D Q_2 \). The definition applies unchanged to unions of conjunctive queries.

DED Implication. We say that DED \( d \) is implied by the set of DEDs \( \Sigma \), denoted \( \Sigma \models d \), if every instance satisfying \( \Sigma \) also satisfies \( d \).
Proposition 4.4.1 The DED

\[
(d) \quad \forall u_1 \forall v_1 \left[ r(u_1, v_1) \rightarrow \bigvee_j \exists v_2 \ s_j(u_1, v_2) \right]
\]

is implied by the set of DEDs \( \Sigma \) if and only if the containment

\[
R(u_1) \leftarrow r(u_1, v_1) \quad \subseteq \Sigma \quad S(u_1) \leftarrow \bigcup_j s_j(u_1, v_2)
\]

holds on all instances satisfying \( \Sigma \).

Here \( \overline{w} \) is shorthand for a tuple \( w_1, \ldots, w_k \) of variables and constants. The proof is omitted, but follows easily from the definitions of query containment and DED satisfaction.

Chase with DEDs. We extend the classical relational chase [BV84b], which is a proof procedure for query containment under embedded dependencies (and by the reducibility property, also for dependency implication). First a bit of notation:

A homomorphism from \( \phi_1 \) into \( \phi_2 \) is a mapping \( h \) from the variables of \( \phi_1 \) into those of \( \phi_2 \) such that (i) for every equality atom \( w = w' \) in \( \phi_1 \), \( h(w) = h(w') \) follows from the equality atoms of \( \phi_2 \) and (ii) for every relational atom \( R(w_1, \ldots, w_l) \) in \( \phi_1 \), there is an atom \( R(v_1, \ldots, v_l) \) in \( \phi_2 \) such that \( v_i = h(w_i) \) follows from the equality atoms of \( \phi_2 \). Given conjunctive queries \( Q_1(x_1, \ldots, x_n) \leftarrow \phi_1(x_1, \ldots, x_n, y_1, \ldots, y_m) \) and \( Q_2(u_1, \ldots, u_n) \leftarrow \phi_1(u_1, \ldots, u_n, v_1, \ldots, v_k) \) (\( \phi_1, \phi_2 \) are conjunctions of relational and equality atoms), a containment mapping from \( Q_1 \) to \( Q_2 \) is a homomorphism \( m \) from \( \phi_1 \) to \( \phi_2 \) such that \( m(u_i) = x_i \) for \( 1 \leq i \leq n \).

Let \( d \) be a DED of general form (4.1), \( Q \) be a conjunctive query and let \( h \) be a homomorphism from \( \phi \) into \( Q \). We say that the chase step of \( Q \) with \( d \) using \( h \) is applicable, if \( h \) allows no extension which is a homomorphism from \( \phi \land \psi \) into \( Q \) for any \( 1 \leq i \leq l \). In this case, the result of applying this chase step is the union of queries \( \bigcup_{i=1}^l Q_i \), where each \( Q_i \) is defined as \( Q \land \psi_i(h(x_1), \ldots, h(x_n), f_{i,1}, \ldots, f_{i,k_i}) \), where the \( f_{i,j} \)'s are fresh variables.

Example. For example, chasing

\[
Q(x, y) \leftarrow a(x, y)
\]

with

\[
\forall u \forall v \ [a(u, v) \rightarrow b(u, v) \lor \exists w \ c(v, w)]
\]
using the homomorphism $\{u \mapsto x, v \mapsto y\}$ results in $Q_b \cup Q_c$ with

$$Q_b(x, y) \leftarrow a(x, y), b(x, y)$$
$$Q_c(x, y) \leftarrow a(x, y), c(y, f)$$

Observe that no chase step with the same DED is applicable to either $Q_b$ or $Q_c$. ●

If we continue applying chase steps to each $Q_i$ (with DEDs from a set $D$), we build a chase tree rooted at $Q$, whose subtrees are the chase trees rooted at the $Q_i$'s. The leaves of the chase tree are conjunctive queries to which no chase step with any DED from $D$ applies. In general, the chase may diverge, thus building an infinite tree, but when it terminates, we define its result to be the set of leaves of the chase tree, denoted $\text{chase}_D(Q)$.

**Theorem 4.4.2** Given conjunctive queries $Q_1, Q_2$ and the set $D$ of DEDs, assume that the chase of $Q_1$ with $D$ terminates. Then we have:

1. $Q_1$ is equivalent under $D$ to the union of the leaves of the chase tree, and
2. $Q_1$ is contained in $Q_2$ under $D$ if and only if for every leaf $L \in \text{chase}_D(Q_1)$, either (i) the equality of two distinct constants is implied by the equalities in $L$, or (ii) there is a containment mapping from $Q_2$ into $L$.

Note that condition (i) deals with the case when $L$ is unsatisfiable, that is it returns the empty answer on all instances. In this case, it is trivially contained in any other query.

The proof is omitted, but it is a straightforward generalization of the classical proof given in [BV84b] for the case of embedded dependencies (recall these are DEDs without disjunction). The key idea used in the proof is the fact that for every leaf $L$ of the chase result, if (i) does not hold, $L$ can be viewed as an instance which satisfies the dependencies $D$.

In fact, we retrieve the result of [BV84b] as a particular case of theorem 4.4.2 by observing that in the absence of disjunction, the chase tree degenerates into what [BV84b] calls a chase sequence, having a single leaf. Another particular case is obtained when our DEDs contain no existentials, and only equalities between a variable and a constant are allowed on the right-hand side of the implication in general form (4.1). Such constraints and the idea of chase tree were introduced in [GM99], in the context of incomplete databases.
4.4.3 The Extended C&B Algorithm

Minimal union of conjunctive queries. Definition 4.2.1 extends straightforwardly to unions of conjunctive queries:

**Definition 4.4.3 (Minimal Union of Conjunctive Queries under Constraints)** Let $Q$ be a union of conjunctive queries $Q = \bigcup_{1 \leq i \leq M} Q_i$ and $D$ be a set of dependencies. We say that $Q$ is $D$-minimal if:

(i) there are no distinct $1 \leq k, l \leq M$ such that $Q_k$ is contained in $Q_l$ under $D$ (we say that the union is reduced), and

(ii) there is no $m$, no distinct relational atoms in $Q_m$’s body $R_j(\overline{x})$ (for $1 \leq j \leq k$ for some $k$) and no conjunctions of equalities $C_j$ (for $1 \leq j \leq k$) such that denoting with $Q_{m,j}$ the query obtained from $Q_m$ by replacing $R_j$ with $C_j$, we have that $\bigcup_{1 \leq i \leq M, i \neq m} Q_i \cup Q_m$ is equivalent to $\bigcup_{1 \leq i \leq M, i \neq m} Q_i \cup \{Q_{m,j}\}$. 

Notice that for $M = 1$, the definition coincides with the definition of minimal conjunctive query.

We repeat here the definition of subqueries of a conjunctive query from section B.1:

**Definition 4.4.4 (Subquery of a conjunctive query)** A conjunctive query $SQ$ is a subquery of a conjunctive query $Q$ if there exists a containment mapping $h$ from $SQ$ into $Q$ such that whenever the image of two distinct atoms $R(x), R(y)$ under $h$ coincides, the conjunction of equalities $x = y$ is implied by the equality atoms in $SQ$.

We extend the definition of subquery to unions of conjunctive queries:

**Definition 4.4.5 (Subquery of a union of conjunctive queries)** $SQ \leftarrow \bigcup_i sq_i$ is a subquery of $Q \leftarrow \bigcup_j q_j$ if for every $i$ there is a $j$ such that $SQ \leftarrow sq_i$ is a subquery of $Q \leftarrow q_j$ in the sense of the above definition.

The Extended (C&B) algorithm. The extended algorithm runs in the same two phases:

Phase 1: the chase. In the first phase, the original query is chased with the available constraints. Assume the chase terminates, and call its result $U$. 

41
Phase 2: the backchase. In the second stage, the subqueries of $U$ are considered as candidates for rewritings of $Q$.

Again, regardless of how we explore the search space of $U$’s subqueries, we have to check their equivalence under all available constraints to the original query. This is done again using the chase, according to theorem 4.4.2.

**Theorem 4.4.6** Let $Q = \bigcup_{1 \leq i \leq M} Q_i$ be a union of conjunctive queries such that the chase of each $Q_i$ with the set of DEDs $D$ terminates, yielding the $m_i$ leaves $L_{i,1}, \ldots, L_{i,m_i}$. Denote $U$ the union of conjunctive queries $U = \bigcup_{1 \leq i \leq M, 1 \leq j \leq m_i} L_{i,j}$.

Then any $D$-minimal rewriting of $Q$ under $D$ is a subquery of $U$.

The proof is shown in appendix B.3.

Notice that there is no explicit mention of views in the theorem. The algorithm is oblivious of where the dependencies come from. As mentioned above, this result generalizes the one given in [DPT99] to a significantly larger class of dependencies, finding rewritings using views under integrity constraints, and even rewritings under integrity constraints in the absence of any views (this being an instance of semantic optimization).

More importantly, it turns out that the kind of dependencies used in TIX and used to compile schema mappings and XML integrity constraints go beyond those allowed in [DPT99] for the following two reasons: on one hand, they allow disjunction but more importantly, they are so-called “non-total” or “non-full” dependencies, by which we mean that they allow existentially quantified variables in the conclusion of the implication. However,

1. they are DEDs and
2. we show in appendix C.4 that the chase with them terminates,

so theorem 4.4.6 applies. This fact turns out to be crucial for the applicability of the C&B method to the problem of XQuery reformulation, as shown in chapter 6. In particular, it turns out that the semantics of Skolem functions and copying of XML elements is captured by such non-total dependencies (see chapter 5).
Chapter 5

Compiling the Schema Correspondence

This chapter describes the compilation of the schema correspondence and XML integrity constraints (Figure 5.1).

We illustrate our compilation procedure on an example. We take for this the mapping MusicMap from Music.rdb to Music.xml and we give some of the TIX constraints that capture it. This mapping is really a composition of the generic encoding of RDB into XML (Figure 2.10) and the actual XQuery in Figure 2.11, call this XQuery MusicMap. We first show how to capture the encoding, then we express MusicMap with constraints,
showing how we address the challenges listed in the introduction.

5.1 Compiling Generic XML Encodings of Relations

These encodings relate every relational tuple with a distinct XML element (see Figure 2.10). This element is uniquely determined by the values of the tuple’s attributes. Therefore, the encoding is really a function from tuples to elements, which is also injective (distinct tuples are encoded as distinct elements).

In our example, there is a unique rdb-element $e_r$ which is a child of the root. Every tuple $song(t,aid)$ is encoded as a song-element $e_s$, which is a child of $e_r$ (author and maiden tuples are similar). $e_s$ in turn has a title-subelement $e_t$ and an aid-subelement $e_a$, which contain as text a copy of the corresponding title, respectively aid attributes. We can model this encoding in two steps: first we state the existence of the unique element $e_r$ as well as of the elements $e_s,e_t,e_a$ for every song tuple, as functions of its attribute values. Then we specify the tag names of these elements, and the parent-child relationship between them.  

```
root
  |
Fr()=er
step1
  |
song(‘‘Alma’’, ‘‘2’’) -----> Fs(Alma,2)=es
    / \
  Ft(Alma,2)=et Fa(Alma,2)=ea

root
  |
tag(er)=’’rdb’’
step2
  |
    ---> tag(es)=’’song’’
      / \
  tag(et)=’’title’’ tag(ea)=’’aid’’
```

In formal XQuery data model terminology, in the first step we actually specify the existence of element nodes, whose tag information and parent-child relationship are given in the second step.

\footnote{In formal XQuery data model terminology, in the first step we actually specify the existence of element nodes, whose tag information and parent-child relationship are given in the second step.}
Step 1: Elements as functions of tuples. Let’s call \(F_s, F_t, F_a\) the functions taking as argument a song tuple and returning an XML element: \(F_s(t, aid) = e_s, F_t(t, aid) = e_t, F_a(t, aid) = e_a\). The unique element \(e_r\) is independent of any tuple and can be modeled as the image of a function of no arguments: \(F_r() = e_r\). We call \(F_s, F_t, F_a, F_r\) Skolem functions, using the terminology from SilkRoute’s mapping language, RXL [FTS00] for similar functions that take as argument relational data and create new XML nodes.

In order to capture these functions with constraints, we extend the schema with the relations \(G_s, G_t, G_a, G_r\) storing the associated function graphs: \(F_K(t, aid) = e_K\) if and only if \(G_K(t, aid, e_K)\) where \(K\) is either of \(s, t, a\), and \(F_r() = e_r\) if and only if \(G_r(e_r)\). For these relations to represent the graphs of the intended Skolem functions, they must satisfy the following constraints: (i) the first two components of each \(G_K\) must functionally determine the third (in other words, the relations are functional), (ii) their projection on the first two components—corresponding to the domain of the Skolem functions—must coincide with the set of song tuples, (iii) the third component must functionally determine the first two (the Skolem functions are injective, as they encode distinct tuples as distinct elements). This is how we express these constraints (where \(G\) stands for \(G_K\) and \(F\) for \(F_K\), for any \(K\)):

\[ G’s	ext{ third component is a function } F\text{ of its first two : } \]
\[ \forall t \forall a \forall e_1 \forall e_2 \left[ G(t, a, e_1) \land G(t, a, e_2) \rightarrow e_1 = e_2 \right] \quad (5.1) \]

\[ \text{song is included in the domain of } F : \]
\[ \forall t \forall a \left[ \text{song}(t, a) \rightarrow \exists n G(t, a, n) \right] \quad (5.2) \]

\[ \text{the domain of } F \text{ is included in song :} \]
\[ \forall t \forall a \forall n \left[ G(t, a, n) \rightarrow \text{song}(t, a) \right] \quad (5.3) \]

\[ F \text{ is injective} \]
\[ \forall t_1 \forall a_1 \forall t_2 \forall a_2 \forall n \left[ G(t_1, a_1, n) \land G(t_2, a_2, n) \rightarrow t_1 = t_2 \land a_1 = a_2 \right] \quad (5.4) \]

\[ ^2 \text{Observe that if we eliminate the existential quantifier in (5.2) above by bringing the statement in Skolem normal form, } n \text{ is expressed as a function } F(t, a), \text{ which explains the naming choice.} \]
The corresponding constraints for \( G_r \) are particular cases of the above, in which there are no arguments \( t, a \) since \( F_r \) takes the empty tuple as argument. It is easy to see that the constraints corresponding to (5.1) and (5.3) above are trivially satisfied in this case, and we are left with the analogons to (5.2) and (5.4), of which we show the latter:

\[
\forall e_1 \forall e_2 \left[ G_r(e_1) \land G_r(e_2) \rightarrow e_1 = e_2 \right] \quad F_r \text{ is injective (} e_r \text{ is unique)} \tag{5.5}
\]

**Step 2: Adding tagged tree structure.** We still have to specify how the elements are organized in the XML tagged tree structure. For example, \( e_r \) is tagged \( \text{rdb} \), \( e_s \) is tagged \( \text{song} \) (shown below), etc.:

\[
\forall t \forall a \forall n \left[ G_s(t, a, n) \rightarrow \text{tag}(n, "song") \right] \tag{5.6}
\]

Moreover, the unique \( e_r \) is a child of the root and every \( e_s \) is a subelement of \( e_r \). Every \( e_t \) is a subelement of the \( e_s \) related to the same tuple, and has a text child whose value corresponds to the \( \text{title} \) attribute of this tuple. We illustrate the constraint capturing the latter statement, as the more complex one. The treatment of \( e_a \) is similar.

\[
\forall t \forall a \forall e_s \forall e_t \left[ G_s(t, a, e_s) \land G_t(t, a, e_t) \rightarrow \text{child}(e_s, e_t) \land \text{text}(e_t, t) \right] \tag{5.7}
\]

Finally, we must state that the encoding contains only information from our music database, and nothing more. That is, any \( \text{rdb, song, title} \) etc. element, appearing anywhere in the document is related to some tuple via a Skolem function. Moreover, its parent is determined by this tuple: the root is the parent of \( e_r \), \( e_r \) that of \( e_s \), and \( e_s \) that of the \( e_t \) related to the same tuple as \( e_s \). Similarly for \( e_a \). We illustrate for \( e_t \):

\[
\forall r \forall d \forall c \left[ \text{root}(r) \land \text{desc}(r, d) \land \text{child}(d, c) \land \text{tag}(c, "title") \rightarrow \exists t \exists a \quad G_s(t, a, d) \land G_t(t, a, c) \right] \tag{5.8}
\]

Let us denote by TIXMusic the set of constraints obtained from the compilation of the encoding as shown above. Notice that all these constraints are disjunctive embedded
dependencies. The size of TIXMusic is easily seen to be linear in the size of the relational schema Music.rdb.

5.2 Compiling mappings/views specified in XQuery

Obstacles in Capturing XQueries with Dependencies. In section 4.1, we have seen how a view defined by a conjunctive query can be expressed using two inclusion dependencies (e.g. \( cv, bv \) on page 32). This technique cannot be directly applied to views defined by XQueries, because these go beyond conjunctive queries in the following ways:

- **XQueries contain descendant navigation steps** (transitive closure of child navigation steps), which cannot be expressed in First Order Logic, in particular in relational conjunctive queries and with constraints.

- **XQueries are interpreted over trees**, while conjunctive queries are interpreted over arbitrary relations. Consequently, even if we encode XML using some form of an edge relation, classical rewriting algorithms will miss even the most obvious rewritings, because they hold only on instances in which the edge relation corresponds to a tree, and not on arbitrary interpretations of this relation.

- XQueries contain nested, correlated subqueries in the return clause. The outer query may contribute to the output even if the inner query is empty. The relationship between input and output must therefore be specified separately for each subquery, while preserving their correlation.

- XQueries create new nodes, which do not exist in the input document, so there is no inclusion relationship between input and output node sets.

- XQueries return deep, recursive copies of elements from the input. If we fail to capture this behavior with dependencies, the algorithm will miss out on reformulations which use a view that copies an XML subtree when the client query navigates into this subtree.

- XQueries have list semantics, as opposed to conjunctive queries, who have set semantics.
We illustrate on MusicMap given in Figure 2.11.

According to the XQuery standard, MusicMap executes in two stages. In the first, which we will call the binding stage, the for clause is evaluated, returning the list of bindings for the variable song (introduced by the $ sign), ordered in the document order. song is bound to every song-element reachable by navigating to a descendant element of the root (note the use of // for descendant navigation), and from there to a child song-element. In the second stage, which we call the tagging stage, a fresh lied element is created and output for every binding obtained in the first stage. This element is a child of the new document’s root. Its contents is given by the result of several nested, correlated queries, whose execution follows in turn the binding and tagging steps. For example, the $song/title subexpression is shorthand for a query returning deep, recursive copies of all child title-elements of the element to which song was bound. Also, the contents of the composer-element is explicitly defined by a nested query. We have highlighted in bold font the ingredients going beyond conjunctive queries, and we show below how we capture them with constraints.

**Obstacle: list semantics for variable bindings.**

**Partial Solution: reduce lists to ordered sets, disregard order.**

Our approach, based on compilation into relational constraints does not fully capture document order. But what are the limitations implied by this? Notice that reformulating queries that order their result according to the values in the input (thus overriding the original document order) is an unrelated, easy problem. The hard issue is that of deciding whether a certain reformulation preserves the ordering in the result of the original query when the latter performs no ordering of its own, using instead the document order. In scenarios involving no native XML documents, but instead publishing relational/OO/LDAP data, there is no document order to begin with, so this becomes a non-issue. For the case of native XML sources, this problem is still an open research issue, which we do not address in this thesis.

Even if we disregard order in the list of variable bindings, we are not yet in a conjunctive query context, because now the list becomes a bag. This bag does not contain duplicates if all variables are bound to elements (two distinct entries in the list of bindings for an
element-typed variable may be isomorphic, but will have distinct node identity because the result of any path expression contains no duplicate node identities [W3Ce]. In other words, the list is really an ordered set in this case. This is the case for the outer for clause of MusicMap in our example and holds in general if no parent navigation is involved, and all variables are bound to elements, rather than to text and attribute values. The same observation holds if variables are bound using the distinct keyword. Otherwise, the following simple trick will help us reduce the bag of bindings to a set: add to the for clause a variable that binds to the element owning the attribute/tag/text (if not already there). For instance, the clause for //lied//maiden/text() $mt in Q from our example is equivalently rewritten to for //lied//maiden $m, $m/text() $mt.

This trick is not needed in the case of MusicMap, and we can therefore define the set of variable bindings in its outermost for clause as the result of the XBind query

$$XB_O(s) \leftarrow [[[\text{song}]](s)$$

which compiles to the simple conjunctive query against schema GReX (as detailed in section A.2):

$$B_O(s) \leftarrow \text{root}(r), \text{desc}(r,d), \text{child}(d,s), \text{tag}(s,\text{"song"})$$

Now we proceed as in the original (relational) C&B method: we extend the schema with a new relation symbol $B_O$, and specify the new relation’s extent with two inclusion constraints; one says that it includes the result of the defining query, the other one gives the inverse inclusion:

\[ \forall r \forall d \forall s \ [\text{root}(r) \land \text{desc}(r,d) \land \text{child}(d,s) \land \text{tag}(s,\text{" lied"}) \rightarrow B_O(s)] \tag{5.9} \]

\[ \forall s \ [B_O(s) \rightarrow \exists r \exists d \text{root}(r) \land \text{desc}(r,d) \land \text{child}(d,s) \land \text{tag}(s,\text{" lied"})] \tag{5.10} \]

\[ \text{This is true for the fragment of XQuery that does not use the concatenation operator ‘,’ which appends the lists yielded by two XPath expressions, preserving duplicates. It can be used in variable bindings such as $v \text{ in p1,p2}$, where p1,p2 may denote lists of nodes with nonempty intersection. In the remainder of this chapter, we will only work with XQueries that do not use the concatenation operator. See chapter 10 for the theory of rewriting queries with bag semantics, applied to SQL. The application of our results to XQuery with concatenation is future work.} \]
Obstacles: interpretation over tagged trees, and descendant navigation in them

Solution: add constraints of TIX

Notice the child, tag and desc atoms in BO’s definition, as well as in the corresponding constraints. Of course, we do not interpret these symbols over arbitrary relations, but rather we want to capture the intended meaning of navigation in the XML tree, to a descendant of the root, and from there to a child with tag “lied”. Recall from section 3.1 that part of the intended meaning is captured using the constraints in TIX (saying for instance that there are no cycles in the graph, that each element has precisely one parent, that the descendant relationship is transitive, reflexive, etc.) Therefore, when performing the reformulation of a query with respect to MusicMap we will do so under the constraints describing MusicMap and the constraints in TIX. It turns out that under certain conditions, the reformulations we obtain this way are precisely all valid reformulations of BO over XML documents (see theorem 6.2.1 below).

Obstacle: nested queries in return clause.

Solution: specify sets of variable bindings separately, using decorrelation.

In the second, tagging stage of MusicMap’s execution, the list of bindings is scanned and for every song-element bound to song, a distinct lied-element is output. The $song/title expression nested within the lied tag is shorthand for a nested query iterating over all title-subelements of the song element, and returning copies thereof:

\[ Q_{lied}(\text{$song$}) = \text{for $t$ in $song/title$ return $t$.} \]

Let’s call this query \( Q_{lied} \), and observe that it is correlated with the outer query part via the song variable (a fact denoted by its appearance in parentheses). We show below how we express \( Q_{lied} \) with constraints. All other nested queries are captured similarly.

Although here we are interested in capturing the semantics of XQueries containing nested subqueries, we will do so by borrowing an idea developed for their evaluation. A naive evaluation strategy suggested by the above semantics specification would execute as many calls of \( Q_{lied} \) as there are distinct bindings for song. However, classical work on optimization of nested SQL queries suggests an alternative strategy, based on decorrelation ([SPL96]). According to this strategy, the bindings of each for clause would be
first computed separately, and then put together using outer joins. Here for instance is an
XBind query defining the set of bindings for \( Q_{\text{l}i\text{ed}} \), where the bindings of \( \text{song} \) are inherited
from outer for clause:

\[
XB_{\text{lied}}(s,t) \leftarrow XB_{O}(s), [/title](s,t)
\]

Now we can use the same compilation algorithm to obtain the conjunctive query

\[
B_{\text{lied}}(s,t) \leftarrow B_{O}(s), \text{child}(s,t), \text{tag}(t, "title")
\]

\( B_{\text{lied}} \) is captured again with two inclusion constraints, just as done for \( B_{O} \). We show
only one inclusion:

\[
\forall s, t \exists ! B_{O}(s) \wedge \text{child}(s,t) \wedge \text{tag}(t, "title") \Rightarrow B_{\text{lied}}(s,t) \quad \text{every variable binding in } Q_{\text{l}i\text{ed}} \text{ is in } B_{O}(s,t)
\]

We proceed similarly for the variables bound by the subqueries nested within the
composer, last and maiden tags.

So far, we have only described the binding stage of an XQuery, specifying the sets of
tuples of variable bindings they compute (illustrated on \( B_{O}, B_{\text{lied}} \)). However, we still have
to describe how these bindings are used for the creation of new elements in the tagging
stage. Note that elements are created in two situations: either as new elements one for
each occurrence of a tag in the return clause, or as copies of existing elements (such as
the title subelement of the \( \text{song} \)-element to which the variable \( \$\text{song} \) was bound. We
first treat the creation of new elements.

**Obstacle: new element creation in return clause.**

**Solution: model it using injective Skolem functions.**

For every binding of the variable \( \text{song} \) in \( Q \)'s outermost for clause, new, distinct lied,
composer, etc. elements are created and output. Since distinct bindings generate distinct
elements, there is a one-to-one correspondence between the tuples of bindings in \( B_{O} \) and
these lied and composer elements in \( Q \)'s output. Recall from section 5.1 that we already
know how to describe the fact that XML elements are uniquely determined by relational
tuples, namely when the latter are encoded as the former. Observe that the only difference
from the encoding scenario is how the tuples related to XML elements are obtained: in the encoding scenario, they were tuples in the original data sources, while here they are tuples of variable bindings that are (conceptually) computed in an XQuery’s binding stage. Regardless of where these tuples come from, we can describe their correspondence to XML elements applying the same idea: introduce Skolem functions and describe their graphs with constraints.

Let $F_{\text{lied}}, F_{\text{comp}}$ be the Skolem functions relating each binding tuple $s$ in $B_O$ to the `lied`, respectively `composer` element generated for $s$. Also, let musicRoot be a function of no arguments whose result is the root of the virtual Music.xml document. Similarly, $F_{\text{xdb}}$ is a Skolem function whose result is the top `xdb` element. Then $Q$’s execution is described below in terms of these Skolem functions. For simplicity of presentation, we omit the nested queries, handled similarly by more nested loops and Skolem functions which take as arguments the tuples of variable bindings of their respective queries.

```plaintext
compute set of tuples of variable bindings $B_O$;

rRoot = musicRoot(); $e_{\text{xdb}} = F_{\text{xdb}}();$

make $e_{\text{xdb}}$ a child of $rRoot$ tagged `'xdb'``;

determine binding $s$ in $B_O$ {

e_{\text{lied}} = F_{\text{lied}}(s); e_{\text{comp}} = F_{\text{comp}}(s);

make $e_{\text{lied}}$ a child of $e_{\text{xdb}}$, tagged `'lied'``;

make $e_{\text{comp}}$ a child of $e_{\text{lied}}$ tagged `'composer'``;

... continue recursively for nested queries ...
}
```

We omit the constraints capturing the Skolem functions, as they have the same shape as constraints (5.1) through (5.4) shown in 5.1. The constraints organizing the created elements in the tagged tree structure specified by the return clause are similar to constraints (5.7) through (5.8).

**Obstacle:** deep copy of XML subtrees in the output.

**Solution:** element copies described by Skolem functions, copies of subtrees by recursive constraints
Recall that the nested query $Q_{lied}$ does not return the title elements bound to by the variable $t$, but rather deep, recursive copies thereof. In our scenario, MusicMap is expressed over the XML encoding of relations and therefore it is possible to conclude that the subtree rooted at title elements only contains a text which is equal to the value of the title attribute in the corresponding song tuple. However, we adopt a more general approach here: we treat the encoded RDB as an arbitrary XML document, call it $X_{encode}(\text{Music.rdb})$, and we show how to capture the copying of arbitrary subtrees of the title elements. It is the job of the reformulation algorithm to arrive at the conclusion that title subtrees are really just text copies of a certain relational attribute, by reasoning with the constraints in TIXMusic. This has the advantage that we handle XQueries with arbitrary input, not just those ranging over XML encoding of relational data. We can therefore capture in the same way other mappings expressed by XQueries, such as materialized XML views of original XML documents.

Since a distinct deep copy is generated for each binding $(s,t)$ in $B_{lied}$, we formalize as the predicate $C_{lied}(s,t,e,e')$ the fact that element $e'$ is the copy determined by $(s,t)$ of a descendant $e$ of the title-element $t$. Notice that for any fixed copy (that is, for fixed $s,t$), the subtree rooted at $t$ and its copy are in one-to-one correspondence: $C_{lied}(s,t,e,e') \iff F_{s,t}(e) = e'$ for some injective function $F_{s,t}$. This correspondence is modeled by constraints saying that for fixed $s,t$, the third component of $C_{lied}$-tuples functionally determines the fourth, and viceversa:

$$
\forall s \forall t \forall e_1 \forall e_2 [C_{lied}(s,t,e_1,e') \land C_{lied}(s,t,e_2,e') \rightarrow e_1' = e_2'] \quad C_{lied}(s,t,e,e') \iff F_{s,t}(e) = e' (5.12)
$$

$$
\forall s \forall t \forall e_1 \forall e' [C_{lied}(s,t,e_1,e') \land C_{lied}(s,t,e_2,e') \rightarrow e_1 = e_2] \quad F_{s,t} \text{ is injective} \quad (5.13)
$$

Finally, we use the following constraints to say that for any binding $s,t$, the copy of the title-element $t$ is made a child of the lied-element created for $s,t$ (5.14) and that copying an element means copying its tag (5.15), attributes and text (omitted, similar to (5.15)) and recursively its child (5.16) and descendant elements (omitted, similar to (5.16)).

$$
\text{copy title, make child of lied}
$$

$$
\forall s \forall t \forall e_1 [F_{lied}(s,e_1) \land B_{lied}(s,t) \rightarrow \exists \ t' C_{lied}(s,t,t') \land \text{child}(e_1,t') \land \text{tag}(t',"title")] \quad (5.14)
$$
Putting all pieces together, we denote the constraints capturing the meaning of MusicMap with MusicMapConstr. It is easy to see that the size of MusicMapConstr is linear in that of MusicMap.

Remark. The constraints in MusicMapConstr show how we capture the semantics of an XQuery (MusicMap) in a purely declarative way, using first-order logic statements (constraints). The benefit of using constraints is that they are more amenable to reasoning with than the algorithmic specification in the W3C recommendation. However, we do not advocate constraints as user-level language. Instead, we translate to them mappings given as queries in the XQuery, SQL, RXL, STORED, etc. languages. The DBA can moreover write his own mappings, possibly more complex than queries in the above languages, as long as they can be expressed with/translated to constraints. Our personal favorite is an RXL-style syntax allowing relational atoms in return clause, or equivalently, XQuery extended with Skolem functions as explicit primitives and with the ability to bind relational tuple variables.

5.3 Mappings Specified Directly with Constraints

Recall from our example the relational view RV.rdb storing the more structured part of the Vienna.xml document. We have expressed this view in XQuery, but there is also a different, more straightforward way to achieve the same effect, writing directly constraints. We capture RV.rdb with a set RViewConstr of two constraints, one for each inclusion between the extent of RV.rdb and the fragment of Vienna.xml it stores. Here is one of them:

\[
\forall r \forall d \forall p \forall pm \forall pn \forall s \forall f \forall t \forall m \forall sm t\forall smt \quad [\text{viennaRoot}(r) \land \text{desc}(r,d) \land \text{child}(d,p) \land \text{tag}(p,"person") \\
\land \text{child}(p,pm) \land \text{tag}(pm,"name") \land \text{text}(pn,pmt) \\
\land \text{child}(p,s) \land \text{tag}(s,"spouse")]
\]
\( \land \text{child}(s, sf) \land \text{tag}(sf, \text{"first"}) \land \text{text}(sf, sft) \)
\( \land \text{child}(s, sm) \land \text{tag}(sm, \text{"maiden"}) \land \text{text}(sm, smt) \)
\( \rightarrow \text{RV}(pnt, sft, smt) \)

Note that the binding part of the XQuery in Figure 2.12 corresponds to the premise of (5.17)'s implication. The “shortcut” in expressing \( RV.rdb \) is taken in the conclusion of the implication, which mentions the target relation, rather than its XML encoding.

### 5.4 Compiling XML Integrity Constraints

We have seen an example for a simple XML integrity constraint (\( \text{SXIC} \)) in section 2.2. The semantics of \( \text{SXICs} \) is readily inferred from the semantics of XPaths and the standard semantics of logical connectives and quantifiers (see section 9.2 for a formal definition).

Combining the \( \text{Path}() \)-translation of the XPath atoms shown in section A.2 with a straightforward translation of logical connectives and quantifiers, we translate \( \text{SXICs} \) into disjunctive embedded dependencies (DEDs) over the schema of TIX: we translate every path atom \( [p](v, w) \) to the conjunction of all goals in \( \text{Path}([p](v, w)) \), existentially quantifying all fresh variables introduced during the translation (of course, when bringing the sentence to Prenex Normal Form, the fresh variables from the premise become universally quantified, while those in the conclusion remain existentially quantified). For example, \( \text{SXIC}(\text{someTitle}) \) from page 24 translates to

\[
\forall r, d, l \ [\text{musicRoot}(r) \land \text{desc}(r, d) \land \text{child}(d, l) \land \text{tag}(l, \text{"lied"})] \rightarrow \exists t \ \text{child}(l, t) \land \text{tag}(t, \text{"title"})
\]

and

\[
\forall x \ [\text{/A/@gender}(x) \rightarrow x = \text{"m"} \lor x = \text{"f"}]
\]

translates to

\[
\forall r, x, u, v \ [\ \text{root}(r) \land \text{desc}(r, u) \land \text{child}(u, v) \land \text{tag}(v, A) \land \text{attr}(v, \text{"gender"}, x) \rightarrow x = \text{"m"} \lor x = \text{"f"}]
\]

where \( \text{root} \) is inferred from the context.
Chapter 6

Putting It All Together

Plans: reformulations using auxiliary schema. If any variables of the XBind query $Xb$ are bound to element nodes, then $Xb$ cannot be reformulated against the storage schema: if the latter is relational, it contains no XML nodes, and if it is mixed, then the node identities in the storage and published data are disjoint. We hence need to find query “plans” which collect data from the storage but also invent and copy nodes, according to the semantics of the XQuery views that define the schema correspondence.

We have shown in section 5.2 how to model this semantics using Skolem and copy functions. Suppose a plan retrieves the storage data tuples that satisfy condition $c(x)$ and returns $y$ and an invented node $n = F(z)$ where $F$ is a Skolem function and $y, z \subseteq x$. This plan can be described as the query $P(y, n) \leftarrow c(x), G(n, z)$, with $G$ the graph of $F$ $(G(n, z) \Leftrightarrow n = F(z))$. Denote with $\text{Aux}$ the relational symbols modeling the graphs of Skolem and copy functions. Then any plan can be represented by a query against the extended storage schema $S \cup \text{Aux}$.

**Algorithm for reformulation of XQuery navigation part**

**Parameters:**
- a schema correspondence described by a set of behaved XQuery views $V$.
- the set $C_X$ of XICs over the various XML documents (public or storage)
- the set $C_R$ of relational integrity constraints over the relational part of the storage schema $S$.

**Input:** a behaved XQuery $Q$ against the public schema.

**Output:** set of reformulation plans for $Q$

**Do:**
0. *configuration:*

Compile the schema correspondence to the set of DEDs \( c(V) \).

In the process, we introduce the set \( \text{aux} \) of Skolem and copy function graphs (see section 5.2).

Compile \( C_X \) to the set \( c(C_X) \) of DEDs.

1. *preprocessing:*

Let \( (X_{bi})_{1 \leq i \leq n} \) be the XBind queries for \( Q \), obtained by decorrelation.

Compile them to unions of conjunctive queries \( c(X_{bi}) \).

2. *reformulation:*

Let \( R_i \) be the set of reformulations against \( S \cup \text{aux} \), obtained by applying the C&B algorithm to \( c(X_{bi}) \) under the DEDs \( TIX \cup c(V) \cup c(C_X) \cup C_R \).

Return only queries that correspond to viable reformulation plans.

Note that the configuration of the MARS system (step 0) is independent of the input and can hence be performed off-line, once and for all. Configuration means compiling the schema correspondence and integrity constraints to DEDs (as detailed in chapter 5).

Steps 1 and 2 are responsible for the reformulation and are performed on-line.

In step 1, the client XQuery is fed to a preprocessor, which, after simple normalization transformations (of the type performed in [MFK01], and not further detailed here), breaks it into one or more decorrelated XBind queries \( XB_i \) \((1 \leq i \leq n)\) and a tagging template. Each XBind query \( X_{bi} \) is compiled relationally over the schema \( GReX \) to a union of conjunctive queries \( c(XB_i) \).

Step 2. Each \( c(XB_i) \) is still expressed against the public, virtual, XML schema and hence it is not directly executable. Instead, it is reformulated by the C&B module, using all available DEDs. Detecting queries that correspond to viable reformulation plans is done as follows: The invention and copying of nodes can be “read off” the atoms from \( \text{aux} \) appearing in a reformulation, thus yielding a reformulation plan. Further, there may be reformulations that correspond to no viable plans, because they disregard the limited binding pattern [LRU96] associated to any relation modeling the graph of a (Skolem or copy) function: the arguments to the function are inputs, the invented node is the output. Such reformulations are easily filtered out.
Recall that all reformulations are found among the subqueries of the universal plan. There are several ways of exploring the space of subqueries. All of them have one operation in common: regardless of which subqueries we explore, we have to check their equivalence under all available constraints to the original query \((B_Q\) in our example) which is done again using the chase, according to theorem C.3.1.

**Cost-based pruning** Due to the redundancy, \(c(XB_i)\) has in general several equivalent reformulations (denoted by the set \(R_i\)), some potentially cheaper to execute than others. The system can enumerate all of these, but, more usefully, the actual implementation provides a plug for a cost module that is called during the backchase stage, implementing a cost-based pruning strategy which avoids the exploration of the entire space of subqueries.

This pruning strategy is borrowed from [Pop00]. It proceeds bottom-up, starting from subqueries of the universal plan \(U\) consisting of one atom, going on to subqueries of two atoms, and so on, until it hits the first subquery \(R\) that is equivalent to \(c(XB_i)\) under all available constraints. \(R\) is a reformulation of \(c(XB_i)\). Compute its cost and continue the exploration, pruning away superqueries of \(R\) as well as all other subqueries of \(U\) whose cost is larger than the best cost found so far. It is easy to see that the obtained reformulation is minimal, because none of its superqueries are explored. If the cost model is *monotonic* (i.e. the cost of a query is no greater than that of any of its superqueries), the best reformulation under the cost model must be a minimal one. Hence cost-based pruning is guaranteed to end up keeping the globally cheapest reformulation. For non-monotonic cost models, this guarantee does not hold, but cost-based pruning remains a useful heuristic.

**Heuristic:** cost-based pruning using number of atoms as cost. The development of a reasonable cost model for XML navigation is still a research issue (see [CJK+01, AAN01] for initial work, which however does not yet cover the full XQuery navigation). We found however that the heuristic of using the number of scans as cost tends to favor reformulations using relational sources rather than XML documents whenever possible. This is because usually relational storage is in-lined [STZ+99], i.e. one single relational tuple contains data that corresponds to several leaves of an XML tree, who can only be reached in several navigation steps.
Further Processing of the chosen reformulation  We only sketch the steps with the chosen reformulation undergoes in order to be answered. First, it is broken into queries sent to the individual data sources, each translated to the language these sources speak, be it XQuery, SQL, LDAP- or DOM-based programs). Each stored source will return a stream of tuples of variable bindings. The streams returned for all R_i’s are combined and turned into XML by a component called the tagger that uses the tagging template. If the amount of data returned by the source queries is large, tagging can become a bottleneck. This issue is orthogonal to the reformulation issue that we investigate in this thesis. Ample research has been conducted on it [FMS01, CKS+00], and in MARS we have adopted the *late tagging, sorted outer union* approach of [CKS+00] because of its simplicity. It is based on the idea of [SPL96] for outer joining the results of the decorrelated queries to restore the result of the original nested query. [FMS01] proposes a more efficient, but more sophisticated approach. We plan to investigate in future work how to integrate this tagging approach with our reformulation strategy.

6.1 An Example

Suppose we want to reformulate query Q from Figure 2.8. We have seen in chapter 5 how the configuration of the algorithm is performed in this case, using compilation to constraints. The preprocessing part was also illustrated: section A.2 shows the translation of the outermost XBind query of Q to the conjunctive query B_Q. The extraction of nested queries was illustrated on page 50 on the query Q_{lied}. We now illustrate the reformulation step. We repeat below the definition of B_Q, as yielded by the compilation process in the presence of the XML integrity constraints (*oneName*) and (*keyName*) on the document Vienna.xml:

\[
B_Q(pnt) \leftarrow viennaRoot(r).desc(r,d).child(d,p).tag(p,"person"),
child(p,pm).tag(pm,"name").text(pm,pnt).child(p,s).tag(s,"spouse"),
child(s,spf).tag(sf,"first").text(sf,sft).child(sm).tag(sm,"maiden"),
text(sm,smt),
musicRoot(r').desc(r,d').child(d',l).tag(l,"lied"),child(l,c).tag(c,"composer"),
\]
Chase. Recall from section 5.3 the constraints $RViewConstr$ capturing the view $RV.rdb$, in particular the constraint (5.17), repeated here:

$$
\forall r \forall d \forall p \forall pn \forall pnt \forall s \forall sf \forall st \forall sm \forall smnt 
| \begin{align*}
& \text{viennaRoot}(r) \land \text{desc}(r, d) \land \text{child}(d, p) \land \text{tag}(p, "person") \\
& \land \text{child}(p, pn) \land \text{tag}(pn, "name") \land \text{text}(pn, pnt) \\
& \land \text{child}(p, s) \land \text{tag}(s, "spouse") \\
& \land \text{child}(s, sf) \land \text{tag}(sf, "first") \land \text{text}(sf, st) \\
& \land \text{child}(s, sm) \land \text{tag}(sm, "maiden") \land \text{text}(sm, smnt) \\
& \rightarrow RV(pnt, st, smt)
\end{align*}
$$

Now observe that the identity mapping on the variables $r, d, p, pn, pnt, s, sf, st, sm, smnt$ makes the premise of the implication in (5.17) a subset of $B_Q$’s atoms, namely the one corresponding to its first three lines. Also, there is no $RV(pnt, st, smt)$-atom in $B_Q$, so the identity mapping cannot be extended to the conclusion of the implication. A chase step applies therefore, and its effect is that of adding the latter atom to those of $B_Q$. We observe that the effect of chasing with the constraints from $RViewConstr$ is that of bringing into the chase result a $RView$-atom. Similarly, by continuing the chase with constraints from $MusicMapConstr \cup TIXMusic \cup TIX$ it follows that $Author$, $Song$ and $Maiden$-atoms are eventually added.

The final result of the chase stage is a large, redundant query $U$, expressed against both the storage and the published schema, and therefore not executable. Figure 6.1 shows part of the chase result, depicted as a graph whose nodes correspond to the variables of $U$. The pair of atoms $\text{child}(x, y), \text{tag}(y, "n")$ is depicted as an arrow whose pointed end is labeled $n$, and the atom $\text{text}(x, y)$ is shown as an arrow whose pointed end is labeled $\text{text}$. Equality atoms are depicted as dashed lines between the nodes corresponding to the involved variables. In order not to clutter the figure, we omit most of the atoms corresponding to the variable binding relations (such as $B_O, B_{tied}$), Skolem functions (such
as $G_s, G_t$), copy functions (such as $C_{\text{lied}}$), and all but one desc atoms between all pairs of reachable XML elements. All of these are actually added during the chase. The only shown desc atom is depicted using an arrow with dotted line.

**Backchase** In this phase, we search for reformulations among the subqueries of $U$. Observe that the queries corresponding to the bindings of $R_1, R_2, R_3$ can be found as subqueries of $U$ (their graphical representation is a subgraph of $U$’s graphical representation).

We can prune the set of subqueries we inspect using domain-specific knowledge. Our pruning techniques are illustrated below and explained in detail in Chapter 7.

**Maximal reformulation** One pruning criterion uses the fact that all reformulations are directly executable, so they must be expressed exclusively in terms of schemas $S \cup \text{Aux}$. Therefore, it is sufficient to look at all subqueries of the largest subquery of $U$ that is induced by keeping only storage schema elements. We call this the *reduced chase result*. In our example, since Music.xml and Xencode(Music.rdb) are virtual documents, their root relations belong neither to the storage schema and therefore nor to the reduced chase result. The latter will therefore contain a subquery $SQ$ consisting of the child, desc, tag, text and attr atoms corresponding to the navigation in the Music.xml and Xencode(Music.rdb) documents. Since the roots were eliminated, $SQ$ cannot contribute to any reformulation, because legal navigation in any XML document (whether virtual or not) can only proceed
by starting from some entry element (typically the root) and performing child and descendant navigation from there. This is why we may safely eliminate $SQ$ from the chase result as well, and in general all subqueries corresponding to XML navigation that isn’t rooted at some valid entry point into the XML tree. We call the remaining subquery the maximal reformulation. What Figure 6.1 really shows is a part of this maximal reformulation.

Minimizing the maximal reformulation Notice that the maximal reformulation is really an executable query (its atoms belong exclusively to the storage schema and all XML navigation is legal). However, it is redundant (it performs the work of $R_1, R_2, R_3$ simultaneously). In scenarios in which we know that the redundancy is expected to be small, and/or when the query involves a single data source whose query optimizer and execution engines are known to be powerful, we may choose to stop here, check equivalence of the maximal reformulation with the original query, and execute the maximal reformulation as is if the answer is positive (otherwise there is no reformulation against the given storage schema). This approach works for the particular case of XPeranto, whose application scenario coincides to the latter case (no redundancy, single source, good optimizer) and the obtained query is simply passed to the relational optimizer. But we may not always be that fortunate: our simple example shows that it is likely that the stored data is spread across several sources, such that no single source optimizer is aware of the other sources and schema mapping involving them. In such cases, we have to minimize the maximal reformulation ourselves in order to minimize access to sources, and, if possible, skip some storage sources altogether (as $R_2$ and $R_3$ do). The benefit of optimizing the maximal reformulation grows if the data sources have only weak optimization and processing capabilities. In such cases, we can follow up with an optional minimization stage.

In this stage, the subqueries of the maximal reformulation are explored, in search for minimal reformulations. Doing so allows us for example to identify $R_2$ as a valid reformulation, rather than joining together the results of $R_1, R_2$ and $R_3$ and accessing all available data sources.

Notice that if we use the heuristic cost model based on the number of atoms, the relational atom $RV.rdb$ allows access to the person name and to that of its spouse using a single atom (cost 1), while accessing the same information in the XML document requires 15 atoms (recall $B_Q$), resulting in a cost of 15. Under this cost, the backchase stage picks
$R_2$ as the best reformulation, and this is likely to be a reasonable choice.

### 6.2 A Completeness Result for our Reformulation Algorithm

**Behaved XQueries.** There are of course XQuery features we cannot compile to dependencies. User-defined functions, aggregates and universally quantified path qualifiers [W3C] are the main examples. In addition to ruling these features out, behaved XQueries satisfy a few more restrictions (see appendix A.1 for the detailed description of this class of XQueries). The main restriction is ruling out navigation to parent and wildcard child (i.e. child of unspecified tag) (more on this counterintuitive restriction shortly). This class is still quite expressive: it allows navigation to ancestor, descendant and child of specified tag; disjunction and path alternation; inequalities; equalities on values (text and attributes) and on node identities. The queries in all of our examples are behaved. In fact, from a practical perspective, the features that we cover are in our experience the most common ones anyway, with the exception of aggregates.

As discussed shortly, even modest relaxation of these restrictions results in incompleteness of reformulation, suggesting that different techniques are needed to completely reformulate XQueries from beyond this class. We emphasize that the soundness of the algorithm holds for any query that is compilable relationally.

Recalling the notation used in the specification of the algorithm (page 56), denote

$$D \overset{\text{def}}{=} TIX \cup c(V) \cup c(C_X) \cup C_R.$$  

**Theorem 6.2.1 (Relative Completeness)** Assume that the chase of $c(Xb_i)$ with $D$ terminates. Then $R$ is a minimal reformulation of $Xb_i$ under the schema correspondence and integrity constraints if and only if $c(R)$ is a minimal reformulation of $c(Xb_i)$ under $D$.

We call theorem 6.2.1 a *relative completeness* theorem because the completeness of the C&B (theorem 4.2.2) guarantees that all minimal reformulations of $c(Xb_i)$ under $D$ will be found, ensuring the completeness of the overall approach.

**Remarks.** In Chapter 9, we show that containment of XBind queries is undecidable in the presence of XICs that make even modest use of unboundedness. From proposition 4.2.3
it follows that no minimization algorithm is complete for unbounded XICs.

Similarly, it follows that even modest use of non-behaved features such as wildcard child navigation results in an incomplete algorithm unless $NP = \Pi_2^P$: in Chapter 9 we show that containment for XBind queries with wildcard child is $\Pi_2^P$-hard even when the queries are disjunction-free and use no ancestor navigation. On the other hand, the C&B gives us a reformulation in NP in the size of these queries.
Chapter 7

Implementation of the MARS System

MARS was implemented starting from the CoDi system for optimizing queries over Collections and Dictionaries using the C&B. The particular C&B implementation in CoDi was evaluated in [Pop00, PDST00], showing that for typical query sizes the approach is practical in a relational context. Typical examples require us to chase with constraints with under 5 atoms in the premise of the implication, and to backchase universal plans performing no more than 15 joins (this is more than some commercial DBMSs support).

It turns out however that the relational queries and constraints obtained from compiling typical-size XML queries and integrity constraints exceed the above practicality bounds by an order of magnitude. Now “typical” involves backchasing universal plans of a few hundred atoms, and the original C&B implementation does not scale well.

In this chapter, we describe two classes of improvements to the CoDi system that were crucial in achieving practicality in a mixed XML and relational context. First, we reimplemented the C&B algorithm from scratch, obtaining significant speedup over the original implementation (section 7.1). This improvement benefits all scenarios where the CoDi system is applicable. Second, we identified and implemented several XML-specific optimization techniques, which achieve additional speedup exploiting the fact that \texttt{desc}, \texttt{child}, etc. relations encode XML (section 7.2).

The experimental evaluation of the resulting MARS system is presented in section 7.3.
7.1 Reimplementing the Chase and Backchase

**Chasing as query evaluation** Note that every chase step consists of matching the premise of a constraint against the query, and adding its conclusion to it after checking that it isn’t already implied by the query. The chase can hence be seen as the execution of a generalized Datalog program on a small database instance corresponding to the query itself: the query atoms are the tuples in the instance. The idea of regarding the chase as generalized Datalog evaluation on a symbolic database was introduced in [Pop00] in order to help the understanding of the chase. In this work, we use the idea to implement the chase. Consequently, we employ standard relational optimization techniques to speed up this evaluation. For example, we compile constraints down to join trees, whose nodes are relational algebra operators. The query atoms are treated as database tuples and pipelined into the join tree, where they undergo selection and joins.

**Example.** Consider the constraint

\[
(c) \quad \forall x \forall y \forall z \forall u \forall v \left[ R(x, y) \land R(y, z) \land S(z, u) \land S(u, v) \rightarrow T(x, v) \right].
\]

We compile (c)’s premise into a join tree JT\(_P\). (c)’s conclusion is pre-compiled to a join tree JT\(_C\) (this is a single node tree containing a scan of the \(T\) relation). This step is done once and for all, off-line.

Assume that the query below arrives:

\[ Q(a, g) \leftarrow R(a, b), R(b, c), R(c, d), S(d, e), S(e, f), S(f, g) \]

We represent \(Q\) internally as a symbolic database instance \(SDI\) consisting of the relations \(R, S\) of extents \{\((a, b), (b, c), (c, d)\)\} respectively \{\((d, e), (e, f), (f, g)\)\}. Figure 7.1 shows the intermediate results obtained at every node when evaluating JT\(_P\) on \(SDI\). The only tuple propagating to the root is \((b, c, d, e, f)\), which corresponds to the matching \(m = \{x \mapsto b, y \mapsto c, z \mapsto d, u \mapsto e, v \mapsto f\}\).

Next, we need to check that \(m\) cannot be extended to a matching \(\overline{m}\) of (c)’s conclusion into the body of \(Q\). The matches that extend to the conclusion are easily computed in bulk using the same idea: compute the semijoin of the result of evaluating JT\(_P\) with that of evaluating JT\(_C\). In this example, the latter is empty, therefore \(m\) has no extension to
Figure 7.1: Computing a match by evaluating $J T_p$ on $SDI$

$(c)$’s conclusion, and the chase step applies. Its effect is adding the tuple $T(m(x), m(v)) = T(b, f)$ to $SDI$. Any subsequent chase step will operate on the extended $SDI$. $\bullet$

We implemented joins as hash-joins, and pushed selections into them. Using such set-oriented processing techniques to “optimize the optimizer” reduced the time to chase tremendously. In one example, the chase of a query corresponding to a rather large XPath had to be interrupted after 12 hours without reaching completion when performed by the CoDi system, but finished in merely 12 seconds in the MARS implementation. $^1$ Clearly, since in the backchase phase we chase subqueries “back” to the universal plan, this phase benefits from the chase speedup as well. In section 7.3, we perform a comparison of the total chase and backchase times required by the two implementations, and measure a speed-up of at least 10, and as much as 60 even on small examples that were previously handled well by CoDi. Of course, the bag of tricks of relational optimization is not exhausted and one can envision using more of them for further speedup.

**Using the storage schema to prune the backchase** Conceptually, during backchase, we explore all subqueries of the chase result $U$ by considering all subsets of its atoms. The chase result will contain atoms pertaining both to the published and storage schema. Given that the reformulations we seek must be executable queries, they will be expressed solely in terms of the storage schema. There is no point therefore to consider subsets of $U$’s atoms that mention any part of the virtual schema. Note that since the number of subqueries explored by the backchase is exponential in the number of atoms in the body of

---

$^1$The interrupted chase sequence was a prefix of the completed one, thus ruling out the possibility of a non-terminating chase finishing due to a bug in the MARS implementation or due to the non-deterministic pick of a different chase sequence.
the universal plan \( U \), ruling out \( n \) of \( U \)'s atoms entails a reduction of the work by a factor exponential in \( n \).

**Example** For query \( Q \) in the previous example, assume that \( R, S \) belong to the public schema and \( S, T \) to the storage schema (public and storage schema are overlapping). The non-discriminating backchase would consider \( 2^7 \) subsets of scans. If we restrict our attention to storage schema elements only, then the backchase explores \( 2^4 \) subsets. •

### 7.2 XML-Specific Optimizations in MARS

**Short-cutting the chase** We usually do not know what the chase result is. However, for particular sets of constraints, we can predict the chase outcome upfront. Doing so allows us to skip the chase entirely and construct its result directly. For example, we observe that the result of chasing a query solely with the (refl),(base) and (trans) built-in TIX dependencies (recall section 3.1) adds to this query those desc atoms missing from the reflexive, transitive closure of the child atoms. Moreover, we can think of the chase as proceeding according to the following conceptual implementation:

```
repeat until no more chase step applies:
   (1) chase first with (refl),(base) and (trans) until termination
   (2) continue the chase with all other dependencies until termination
end
```

Since we know upfront what the result of phase (1) in every iteration is, (namely the reflexive, transitive closure) we do not have to use the chase to compute it. Instead, we jump directly to the beginning of phase (2) by computing the closure using a traditional adjacency-based algorithm. It turns out that this trick cuts the time to chase considerably.

**Example** Consider a chain of \( n + 1 \) atoms

\[
\text{root}(x_1), \text{child}(x_1, x_2), \text{child}(x_2, x_3), \ldots, \text{child}(x_{n-1}, x_n)
\]  

(7.1)

Chasing with (refl),(base),(trans) will add atoms \( \text{desc}(x_2, x_2), \text{desc}(x_2, x_3), \text{desc}(x_2, x_4) \) etc., resulting in a chase sequence of length \( \frac{n(n+1)}{2} \). Its effect can be simulated by directly computing the transitive closure. •

Recall the experiment mentioned on page 67, in which the time to chase was cut from
over 12 hours to 12 seconds using the new implementation with join trees. If we add this shortcut technique, the same result is obtained in 4 seconds.

**Pruning the Backchase: Redundant desc atoms** Assume that the universal plan $U$ contains both an atom $\text{child}(x, y)$ and an atom $\text{desc}(x, y)$, thus effectively returning all pairs of nodes that are both in a parent-child and an ancestor-descendant relationship. Clearly, by removing $\text{desc}(x, y)$ we preserve equivalence to the original query, and also we do not lose optimality of the reformulation, since in any reasonable cost model accessing the descendants of a node is at least as expensive as accessing its children. We therefore eliminate (before starting the backchase) all $\text{desc}$ atoms that are “parallel” to a chain of $\text{child}$ and $\text{desc}$ atoms. Note that if we don’t, the backchase will do it for us, but it would have to consider a larger space of subsets.

For the example chain (7.1), this means throwing out all $\frac{n(n+1)}{2}$ $\text{desc}$ atoms introduced in the chase phase, resulting in a reduction of the search space by a factor of $O(2^n)$. Note that during the chase, we add these scans, and before the backchase we remove them. One may ask why we can’t avoid adding them in the first place. During the chase, these $\text{desc}$ atoms are needed to enable the chase with other constraints, which mention $\text{desc}$ atoms in their premise.

**Further Pruning of the backchase: legal XQuery navigation** Given the fact that the $\text{child}$, $\text{desc}$, $\text{tag}$ atoms appearing in the chase result $U$ stem from the compilation of XQueries, we can use XQuery-specific knowledge to prune the subsets of atoms that do not correspond to legal XQuery navigation.

- For example, a $\text{tag}$ or $\text{attr}$ atom cannot appear without a corresponding $\text{child}$ or $\text{desc}$ atom. In other words, it is not possible to scan the $\text{tag}$ relation in order to get all element node identities.

- Child and descendant navigation steps must be contiguous. For example, we won’t consider the subquery $\text{root}(x_1), \text{child}(x_2, x_3)$, of the chain (7.1) because it involves jumping directly from $x_1$ to some element $x_2$, instead of navigating there via the missing $\text{child}(x_1, x_2)$ atom. This pruning policy is very effective: in its absence, we
would consider $2^{n+1}$ subsets of scans of the chain. Using it, we only have to look at $O(n^2)$ of them. Notice the drop in quantity from exponential to polynomial.

Of course, a naive implementation in which we first generate the subquery and then check it corresponds to contiguous navigation still results in exponential work. What we do instead is construct a reachability graph of the atoms (the nodes represent atoms, and there is an edge from an atom $a_1$ to atom $a_2$ iff the second variable of $a_1$ coincides with the first of $a_2$. The roots of the graph are given by root atoms.) and traverse this graph in order to generate the legal subsets of atoms. ²

- Similarly, subsets of atoms that do not contain at least an atom describing the root of a document, or some other valid entry point into it, are not considered. This is because XQuery navigation always starts from the root of the document. For example, there is no point considering the subquery $\text{child}(x_1, x_2), \text{child}(x_2, x_3)$ of the chain (7.1). This reduces the number of eligible subqueries further, from $O(n^2)$ to $n + 1$.

### 7.3 Experimental Evaluation

**Platform** The MARS system is implemented in JDK1.2, and the experiments were run on a Dell desktop with a 1.2Ghz PentiumIII processor and 512MB of RAM, running RedHat Linux 6.0.

#### 7.3.1 Speedup of new C&B implementation

This section tries to isolate the speedup we gained over the original C&B implementation by reimplementing the chase and backchase (XML-specific optimizations play no role here). Since the original implementation does not scale well for input sizes that are typical for XML reformulation, we will measure the speedup only on input instances which the original implementation is already handling well.

[PDST00] introduces an experimental configuration used to measure the time to rewrite a relational query using views in the presence of integrity constraints. In order to see how

²This technique is related to that of *sideways information passing* used in System R’s optimizer [SAC+79] to avoid plans that compute Cartesian products (such as $\text{root}(x_1), \text{child}(x_2, x_3)$).
this time evolves as a function of the size of the schema/query/views/constraints, the
configuration is parameterized by \( N_R, N_S, N_V \) as follows:

The public schema contains relations \( R_i(k, f, a_1, a_2, \ldots) \) and \( S_{ij}(a_i, b) \) \((1 \leq i \leq N_R, 1 \leq j \leq N_S)\). \( R_i \)'s attributes \( a_i \) are foreign keys referencing \( a_i \) in \( S_{i,j} \), and \( k \) is a key for \( R \). Moreover, \( R_i \)'s attribute \( f \) is a foreign key referencing \( R_{i+1} \)'s attribute \( k \). Think of \( R_i, S_{ij} \) as storing one large conceptual relation \( U_i \) that has been normalized for storage efficiency.

The query \( Q \) to reformulate is shaped like a chain of \( N_R \) stars, star \( i \) having \( R_i \) as the hub and \( S_{ij} \) as the corners. The relations are joined along the foreign keys: each \( R_i \) is with each \( S_{ij} \) according to \( R_i.a_i = S_{ij}.a_i \), and with \( R_{i+1} \) according to \( R_i.f = R_{i+1}.k \). The query selects the \( b \) attributes of all corner relations it joins.

The storage schema contains the public schema, as well as the materialized views \( V_{i,l} \) \((1 \leq l \leq N_V \leq N_S)\) which join the hub of star \( i \) \((R_i)\) with two if its corners \((S_l \text{ and } S_{l+1})\). The joins are along the foreign keys again, and the view selects the \( b \) attributes of both corner relations, as well as \( R_i \)'s attribute \( k \).

The point of this family of configurations is to illustrate the fact that in the absence of integrity constraints, there is no reformulation using the views, but if we take into account the key constraints on each \( R_i \), then each star join can be reformulated using any subset of the views corresponding to it. There are hence \( 2^{N_V \times N_R} \) possible reformulations, all of
whom are discovered by the C&B algorithm. [PDST00] shows the running times needed by various refinements of the C&B to obtain all reformulations. [Pop00] shows how the running time can be reduced by one order of magnitude using cost-based pruning during the backchase.

In Figure 7.2, we compare the running times of the (unrefined) C&B algorithm in the CoDi and MARS implementations. Every data point corresponds to a value for the $(N_R, N_S, N_V)$ parameters. What is displayed is the ratio of CoDi execution time/MARS execution time. We picked only those configurations which the CoDi unrefined implementation handles well (there are others for which this implementation does not scale, but where the additional refinements and cost-based pruning yield practical running times). Notice that the speedup ranges between 15 and 57.

**Cost-based pruning** For the experimental configuration $(N_R = 2, N_S = 3, N_V = 2)$, the unrefined CoDi implementation takes 129 seconds to enumerate all reformulations (8 of them). The CoDi refinement using cost-based pruning needs only 12.5 seconds to find the best reformulation according to a standard cost model. In contrast, the unrefined MARS implementation needs 2.28 seconds to enumerate the 8 reformulations, and 7.4 seconds to find the best one according to the same cost model. It is interesting to notice that for this example, the bottleneck is not the chase and backchase, but rather the cost estimator. This performs dynamic programming to find the best join reordering for any given subquery and runs in time exponential in the size of the subquery [Pop00].

### 7.3.2 Feasibility of MARS Optimizations

In this section, we show that the MARS Optimizer finds optimal reformulations in time that is only a fraction of the query execution time.

**Configuration based on XML Benchmark** We investigate the behavior of the MARS system in a likely scenario: when the storage is mixed, but mostly relational, and only the highly unstructured XML data is stored as character large objects (clobs) reachable from some tuples via unique identifiers.

**The public schema.** This mixed data is published as an XML document conforming to the DTD of the XML Benchmark project [BCF⁺01]. This project offers a DTD, a set of
20 queries and an XML document generator, intended to measure XQuery execution time for XQuery engines. Since our focus is on query translation and minimization time, we ignore the data generator and use only the queries and the schema. This schema can be viewed at [Pro01a] and it corresponds to the scenario we consider most likely: it consists of a large, highly structured part, augmented with some unstructured XML information. The schema models entities of an auction application, in which various items are bid for (in auctions that are still open), or bought (appearing in the result of closed auctions). Information is also kept on the sellers, bidders and eventual buyers, who are all persons with standard attached information such as address, name, email, credit card, etc., all of it being perfectly structured. The information on auctioned items is both structured (name, provenance, seller, etc.) and unstructured (a description and various annotations containing an arbitrary mix of text, arbitrarily nested within bold and emphatic qualifiers).

The mixed storage schema. The structured part of the data is sufficiently clear from the DTD to allow us to pick a common-sense relational schema, without needing to first perform any involved analysis of what relational storage to choose (such as shown in [DFS99, STZ99]). We simply store such well-structured entities as person, item, open auction, closed auction, etc. in a total of 7 relations, and keep the less structured descriptions and annotations as clob (containing XML text), referenced from these relations.

The queries. The 20 XQueries are designed to exercise those features of the language whose execution is non-trivial, and can be viewed at [Pro01b]. In our graphs, we will refer to them using their official names, $Q_1$ through $Q_{20}$.

The redundancy. We introduce redundancy according to an idealized scenario, in which the system is tuned to answer these 20 queries as fast as possible. To do so, we materialize 20 views, each containing the tuples of variable bindings needed by one of the 20 queries. Since these views can be seen as access support relations in a broader sense, we shall call them ASR$_1$ through ASR$_{20}$. In their presence, each query may be answered by simply scanning its ASR and streaming the tuples to the tagger.

The mapping. It turns out that this relatively simple scenario is already inexpressible by existing systems. As discussed previously, even in the absence of the redundant ASRs,

\footnote{Of course this is a best-case scenario for query execution, but note that from the point of view of reformulation, detecting which ASR is relevant and finding the reformulation is just as hard if the ASR covers all of the query’s navigation or not.}
the fact that the description and annotation data is stored as XML rather than relationally takes us beyond systems like SilkRoute and XPERANTO. Agora cannot be used either, because the storage schema contains a few attributes hidden in the published schema. They are internal keys and foreign keys, introduced for normalization purposes to preserve the relationships such as between items and the XML clobs containing their description, etc. The mapping is therefore expressed as follows: the proprietary relational data is mapped to the published schema via an XQuery. The XML descriptions and annotations are mapped from the published schema to the storage using constraints. The redundant ASRs are described by SQL views over the published schema (seen in the generic relational encoding of XML). All of these compile to about 200 constraints (!) which goes beyond typical relational scenarios, motivating the need to reimplement the C&B to cope with such complexity.

The experiments. We measured the reformulation times for each query, both in the absence of ASRs and in their presence. In the latter case, the maximal reformulation is redundant, joining some proprietary relations with the ASR. The maximal reformulation is minimized according to the backchase stage of our algorithm, by exploring the subqueries bottom-up. We obtain for each query two minimal reformulations, one that uses the ASR and one that does not. The time to find the first reformulation coincides with the time to explore the entire search space using cost-based pruning, when the cost is the number of atoms in the query. In order to show how much more the search would go on without pruning, we also show the time to fully explore the search space (during which we also obtain the other plan).

Conclusions Notice that the time for reformulation is the same whether we add the redundant ASRs or not. Time penalty is incurred only when we switch on the minimization stage. The most time was spent reformulating query \( Q_{10} \), which is highly nested (it has 10 correlated subqueries in the \texttt{return} clause).

The measured times should be considered in light of the time needed to execute queries of this complexity. According to [FMS01], typical query execution times for realistically-sized documents published from databases via mappings that are no more complex than the present one, can last up to 100 seconds. The times we measured are therefore within practicality bounds. As for the execution of \( Q_{10} \), the benchmark record so far is held by...
Figure 7.3: Time for reformulation without minimization

Figure 7.4: Time for reformulation with minimization

to maximal reformulation

the MONET native XML query engine [BCF+01], and it measures 25 minutes.

A heuristic we employ to reduce the search space during minimization, is cost-based pruning using the number of atoms in the query as cost. Consequently, the first plan found is always the one using the ASR, since it contains a single atom. The time for finding this first plan is shown to be reasonable as well (no more than 3 seconds). In general, this heuristic choice of cost gives preference to plans joining as few relations as possible (one cost unit per relation), over plans that perform XML navigation. This is because reaching selected XML fragments typically means performing several navigation steps, where each navigation step contributes a unit to the cost, while the relational storage of the same information is usually in-lined [STZ+99] allowing direct access through the tuple’s attributes.

7.3.3 Benefit of MARS Optimizations

This section addresses the question whether the MARS optimization is worthwhile, i.e. whether we gain anything by first optimizing a query and then executing it, as opposed to executing the query as is.

Configuration based on Star XML Queries and Views In the XML Benchmark configuration, the complexity lies in the number of constraints obtained by compiling the non-trivial mapping. However, there are only two possible reformulations for each query.
In this experiment, we want to stress the backchase implementation, forcing it to pick from exponentially many reformulations. Here is the configuration we used to obtain such a worst-case scenario.

The public schema is the XML equivalent of the schema used in section 7.3.1, for \( N_R = 1, N_S = N_V \) (only one star, and as many views as possible per star).

Every tuple of relation \( R_1(k, a_1, a_2, \ldots) \) is encoded by \( R \)-elements (children of the root element) with \( k-, a_1-, \) subelements, etc. Similarly for the \( S_{1,j} \) relations. The following XIC specifies the XML key constraint on \( R \)-elements, corresponding to the relational key constraint on the \( k \) attribute of \( R_1 \):

\[
\forall r_1 \forall r_2 \forall t [/.//R](r_1) \land [.//k[text()](r_1,t) \land [.//R](r_2) \land [.//k[text()](r_2,t) \rightarrow r_1 = r_2
\]

The storage schema contains the public schema as well as the redundantly materialized XML views corresponding to \( V_{1l} \) from the relational scenario. For example, \( V_{1,2} \) is defined as

```xml
<V>
{let $doc := document("data.xml")
for $r in $doc//R, $rk in $r/K,
   $ra2 in $r/A2, $ra3 in $r/A3,
   $s2 in $doc//S2, $sa2 in $s2/A, $sb2 in $s2/B,
   $s3 in $doc//S3, $sa3 in $s3/A, $sb3 in $s3/B
where
   $ra2/text()=$sa2/text() and $ra3/text()=$sa3/text()
return
   <found>
      <K>{$rk/text()}</K><C>{$sb2/text()}</C><D>{$sb3/text()}</D>
   </found>
}
</V>
```

Note that, since \( V_{1,l} \) and \( V_{1,l+1} \) overlap, there is considerable redundancy in the storage.

The query (family) is the XML equivalent of the relational star query from section 7.3.1. It has a similar shape as the queries defining the views, but it joins \( R \)-elements
with all \( N_S \) \( S_{1,3} \) elements. Consequently, the universal plan will contain all \( N_S \) views, and the backchase stage will have to explore \( 2^{N_S} \) subqueries.

![Graph showing the benefit of MARS optimization](image)

**Figure 7.5: Benefit of MARS optimization**

**What we measure** In Figure 7.5, we show for various values of the \( N_S \) parameter the benefit of performing the MARS optimization, expressed as the difference between the time to execute the star query as is, and the time to find the best reformulation and then execute it. Here “best” is considered with respect to the same simple heuristic cost model used in the XML benchmark scenario. The XQuery engine used is the Galax implementation [FSWC] (the February 2002 version). The dashed lines are used when the target point is too high above 600 seconds.

**Conclusions** Note that the benefit of optimization increases with increasing size of the XML document. This is expected, since the optimization time is independent of it, and only execution time is affected by the size. Whenever the optimized query executes even just a little faster than the original one, one could in principle make the benefit tend to infinity just by increasing the size of the document sufficiently. Notice however that the documents we use are toy examples, containing no more than 200 elements.

Note also that, for a fixed document size, the benefit increases with increasing complexity of the configuration (recall that \( N_S \) determines both the size of the query and the
number of views). The increase is exponential.
Chapter 8

Schema Abstraction

8.1 Accelerating the C&B by Schema Abstraction

We present a strategy for speeding up the C&B algorithm when applied to XML query reformulation. The strategy reduces the number of atoms in the query and constraints, thus obtaining a faster chase, a smaller universal plan, and consequently a faster backchase stage. The basic idea is to abstract a set of atoms corresponding to an XML navigation pattern into a single scan over a virtual relation from an intermediate schema.

For example, if we know that the relatively complex XML tree pattern in Figure 8.1 represents author entities, we can abstract these as tuples of schema given in Figure 8.2.

Figure 8.1: XML representation of author entities

Figure 8.2: relational abstraction of author entities

Note that the identities of internal nodes of the tree (name, address), as well as the parent-child relationships are abstracted away in this relational view. The id attribute holds the identity of the author nodes, and pid that of their parents. These are needed in translating queries against the XML schema to relational queries against the abstraction.
Now consider the following XBind query returning the names of authors who live in a city where a publisher is located:

\[
Xb(1) : = [//author](id), [.//name/last/text()](id,l), \\
[./address//city/text()](id,c), \\
//publisher/address//city/text()](c)
\]

Assume the existence of a relational materialized view associating author last names with the city they live in:

\[
V(l,c) : = [//author](id), [.//name/last/text()](id,l), \\
[./address//city/text()](id,c)
\]

The C&B algorithm will find a reformulation of \(Xb\) using \(V\). It will do so by chasing the relational compilation of \(Xb\) (which we shall denote \(c(Xb)\)) with the constraints capturing \(V\). Here is one of them:

\[
\forall r^d \forall d^i \forall n^x \forall x^y \forall c \\left[ \text{root}(r) \land \text{desc}(r,d) \land \text{child}(d,i) \land \text{tag}(i, 'author') \land \\
\text{child}(id,n) \land \text{tag}(n, 'name') \land \text{child}(n,x) \land \text{tag}(x, 'last') \land \text{text}(x,l) \land \\
\text{child}(id,a) \land \text{tag}(a, 'address') \land \text{child}(a,y) \land \text{tag}(y, 'city') \land \text{text}(y,c) \right] \\
\rightarrow V(l,c) \] (8.1)

The verbosity of this constraint makes for an expensive chase step, since any such step involves matching the premise of the implication against the body of \(c(Xb)\) and the time to match depends on the size (number of atoms) of the premise. The dependence is exponential in the worst case, but in most common scenarios in practice it is polynomial.

The premise size can be reduced by using the above relational abstraction of \(author\) elements. (8.1) turns into (8.2) below, in which we have reduced the number of atoms in the premise by 13:

\[
\forall d^i \forall d^i \forall f^l \forall n^x \forall c^z \forall x^y \forall c^z \left[ \text{Author}(id,d,f,l,str,c,sta,z) \rightarrow V(l,c) \right] \] (8.2)
Similar translations hold for $c(X_b)$. Since the internal tree structure of the XML representation is hidden by the abstraction, we shall call this abstraction technique **schema abstraction**. The application of the C&B algorithm to the abstracted query and constraints holds the potential of greatly improved running time.

**Approach.** Figure 8.3 shows our abstraction strategy. As before, the schema correspondence is compiled to the set $\Delta$ of constraints. We again start by **compiling** the XBInd query $X_b$ to a relational conjunctive query $CQ$ over schema $GReX(P)$, which is the generic relational encoding of XML schema $P$. However, instead of applying the C&B algorithm under $\Delta$ to directly obtain a reformulation $R$, we first **abstract** $CQ$. Denoting with $I$ the intermediate relational schema containing the abstractions corresponding to $P$, abstracting $CQ$ means translating it to a query $CQ'$ against the combined schemas $GReX(P)$ and $I$. This is because the XML navigation over unabstracted data cannot be expressed using $I$. We next **reformulate** $CQ'$ to a query $R'$ over $GReX(S), J$ using the C&B with the

abstracted versions of the constraints in $\Delta$. Here $J$ contains the relational abstractions (if any) of the XML portion of $S$. Notice that, while the abstraction of the query must be done on-line, the abstraction of the constraints $\Delta$ can be done once and for all, off-line. Also notice that this involves reformulation of constraints, which is an area still under research. For now, we assume that this reformulation is done manually, by the database administrator. Automatic reformulation of constraints is the focus of future work. Finally, $R'$ is post-processed by **refining**, which means substituting the relational abstractions from $J$ with the original XML entities from $S$ (encoded relationally in $GReX(S)$).

**Example.** For the above example, **author** belongs to schema $I$, and the abstraction of $CQ$ is

![Diagram](image.png)

Figure 8.3: The overall approach in schema abstraction
The C&B reformulation of $CQ'$ involves chasing with constraint (8.2), which is the abstracted version of (8.1). We obtain, among others, a reformulation using $V$.

**Specifying the abstractions using mappings.** The abstractions can be inferred automatically according to approaches like the one of STORED [DFS99] or hybrid inlining [STZ+99] or specified explicitly by domain experts. The abstraction in the above example would be found by hybrid inlining using a DTD or XML Schema associated with the document. Alternatively, a domain expert could specify the inlining by describing the corresponding XML-to-relational mapping. This can be done using various syntaxes. Here is an example using XBind syntax (see [DFS99] for an alternate syntax of mappings).

```prolog
Author(id,pid,f,l,str,c,sta,z) :-
    [//.](pid), [./author](pid,id),
    [./name/first/text()](id,f), [./name/last/text()](id,l),
    [./address/street/text()](id,str), [./address/city/text()](id,c),
    [./address/state/text()](id,sta), [./address/zip/text()](id,z)
```

**Restrictions on the abstraction mappings to ensure efficient abstraction.** Abstracting a query means reformulating it according to the abstraction mappings. This can be done in two ways. In one approach we could use independent techniques. For example, if the abstraction mappings are the result of running hybrid inlining an a DTD, then the query reformulation algorithm from [STZ+99] could be used. A more desirable approach is to reuse the C&B algorithm. Since the abstractions can be expressed as mappings, we can compile them to constraints and use the C&B reformulation to obtain the abstracted queries. However, if the abstraction mappings are arbitrary, this approach shifts the complexity from the C&B reformulation to the abstraction step itself, thus defeating its purpose.
We identify reasonable restrictions on the expressive power of abstraction mappings which lead to polynomial running time of the C&B when used for abstraction.

1. Abstraction mappings are expressible using XBind syntax. The atoms in the query body contain restricted XPath expressions conforming to the grammar (where \( l \) stands for any constant label)

\[
\begin{align*}
\text{path} & \rightarrow / p | / / p | p \\
p & \rightarrow \text{ } \text{ } l | @l | \text{text()} | . | p_1 / p_2 | p_1 / / p_2
\end{align*}
\]

2. We require that the body of each XBind query be a **tree pattern**, i.e. each variable may appear at most once as the second component of some atom. We will use the notion of **subtree pattern** of a tree pattern in the expected sense.

3. For any two XBind queries \( Xb_1, Xb_2 \) in the mapping, let \( p_1, p_2 \) be root-leaf paths in the their corresponding tree patterns. Then \( p_1 \) is not a prefix of \( p_2 \) and viceversa.

4. A tree pattern \( TP \) may appear in the body of an abstraction mapping only if it is guaranteed (by the DTD or XML Schema) that for every subtree pattern \( SP \) of \( TP \) with the same root as \( TP \), and for every variable binding \( h \) that satisfies \( SP \), \( h \) can be uniquely extended to a variable binding satisfying \( TP \).

The example XBind mapping for the **Author** abstraction satisfies restrictions (1) and (2), and depending on the DTD, it is quite plausible that it satisfy (4) as well. (3) is vacuously satisfied as there is no other XBind query specifying an additional abstraction.

**Proposition 8.1.1** Given an abstraction mapping satisfying the restrictions above, there is a compilation of this mapping to a set of constraints \( \rho \) such that the C&B algorithm applied with \( \rho \) will reformulate (abstract) any query in PTIME in its size.

Restrictions (1) through (4) still allow interesting abstraction mappings. It turns out that the abstraction mapping we would obtain by hybrid inlining satisfies them. This suggests borrowing the hybrid inlining technique (developed for storing the XML data) in order to abstract XML data relationally.
Corollary 8.1.2 If the abstraction mappings correspond to the mapping discovered by hybrid inlining, the abstraction step can be performed in PTIME in the size of the query using the C&B algorithm.

8.2 Benefit of Schema Abstraction

In this experiment, we use the same family of public schema and query as introduced in the Star XML scenario. However, the storage schema contains only the views now, and we measure the ratio of the times to find all reformulations without and with abstraction. This ratio is again measured as a function of the $N_S$ parameter, shown in Figure 8.4. Note the exponential increase of the benefit. The benefit is broken down for (1) the time to obtain a redundant reformulation using all views, (2) the time to perform backchase minimization on this reformulation, to obtain all minimal reformulations, and (3) the total time for finding the redundant reformulation and minimizing it. Recall that the redundant reformulation obtained in (1) is computed by first chasing the query to the universal plan, and then picking the largest subquery that mentions only storage schema elements (in this case views).
To see that the optimization time is quite acceptable when compared to the time to execute the unoptimized query, note that the actual running time for the C&B with abstraction is 0.15 sec (1.1 sec) for $N_S = 3$ ($N_S = 9$), while the running time for the unoptimized query is 0.2 sec (over 10 minutes).
Chapter 9

Containment and Integrity

Constraints for XPath Fragments

We have a general interest in the algorithmic foundations of XML query optimization. The core problem considered here is query equivalence (more generally—query containment) under integrity constraints. There is a large body of research on using constraints in query optimization in traditional databases. Such results do not apply directly to XML queries because of the transitive closure (Kleene star) operator in path expressions, which is not first-order expressible. Significant work that does handle the Kleene star operator has been done on containment of semistructured queries [FLS98, CGL99, CGLV00a]. But these results do not apply directly here because the XML models are trees rather than arbitrary graphs. Nonetheless, the present work benefits from ideas introduced in all these previous papers.

Integrity constraints are a fundamental mechanism for semantic specification in traditional databases. For XML, the design of specification formalisms for integrity constraints is still an ongoing effort, from DTDs [W3Ca], to recent work on keys [BDF+01] and database-style constraints [FS00, FL01] and the current XML Schema standardization effort [W3Cc].

Several of these formalisms build on the XPath standard [W3Cb] or on closely related languages. XPath is also central to XML transformation and query language standards
(XSLT [W3Cl], respectively XQuery [W3Ce]). Consequently, understanding the foundations of XPath query optimization is an important step in tackling the more general problem of XML query optimization.

The XPath standard allows expressions that are strictly more expressive than the XPath expressions supported by the XQuery standard [W3Ce]. However, since XPath is an independent standard, and there is still some debate on which of its features will be adopted by the XQuery standard, we study the more general problem of containment of expressions as defined by the XPath 1.0 standard.

Our core formalism consists of a fragment of XPath that we call simple and which is contained in the XPath fragment allowed in well-behaved XQueries. We introduce a corresponding class of integrity constraints that we call simple XPath integrity constraints (SXIC). SXIC’s can express many database-style constraints, including key and foreign key constraints specified in the XML Schema standard proposal, as well as many constraints implied by DTDs. We identify a subclass of bounded SXIC’s under which containment of simple XPath expressions is decidable, but we show that even modest use of unbounded SXIC’s makes the problem undecidable. In particular, the addition of (unbounded) constraints implied by DTDs leads to undecidability.

We give tight $\Pi^p_2$ bounds for the simple XPath containment problem and tight NP bounds for the disjunction-free subfragment, while even identifying a PTIME subcase. We also show that decidability of containment under SXIC’s still holds if the expressions contain certain additional features (e.g., wildcard) although the complexity jumps to $\Pi^p_2$ even for the disjunction-free subfragment.

We know that our results can be extended to some but not all of the XPath features that depend on document order. The decidability of containment of simple XPath expressions in the presence of DTDs only remains open (although we can show that the problem is PSPACE-hard) as well as the problem for full-fledged XPath expressions, even in the absence of integrity constraints.

Implications for the complexity of XQuery containment It turns out that simple XPaths and some of the extensions we consider are expressible by well-behaved XQueries. Therefore, the lower bounds derived for containment of the former carry over to containment of the latter. The extension of simple XPaths with the wildcard child navigation...
is not expressible by well-behaved XQueries, and its surprising behavior of raising complexity to \( \Pi_2^p \) proves that our XQuery reformulation algorithm cannot be complete for the corresponding XQueries (unless \( NP = \Pi_2^p \)).

### 9.1 Examples

Here is an example of XPath expression

\[
P \overset{\text{def}}{=} ///(a|b)/c[@m = "0" \text{ and } .//d \text{ and } . = /.[@o]/].//@n
\]

For the meaning of this notation see appendix A. There is much more about XPath and its semantics that can be found in [W3Cb, Wad99]. Additional operators are described below. According to the specification, \( P \) above returns the set of attributes named \( n \) of all elements tagged \( c \) which are children of an element tagged \( a \) or \( b \) and have an attribute named \( m \) of value "0", a descendant element tagged \( d \) and some ancestor with an attribute named \( o \).

We also consider a class of XML integrity constraints that combine the logical shape of the classical relational dependencies [AHV95] with a controlled amount of disjunction and with atoms defined by the XPath expressions themselves. Here are a few examples of constraints

\[
\begin{align*}
\text{(oneAddress)} & \quad \forall x \forall s_1 \forall s_2 \quad //\text{person}(x) \wedge //\text{address}(x, s_1) \wedge //\text{address}(x, s_2) \rightarrow s_1 = s_2 \\
\text{(someAddress)} & \quad \forall x \quad //\text{person}(x) \rightarrow \exists y \quad //\text{address}(x, y) \\
\text{(idref)} & \quad \forall x \quad //\text{person}/@\text{spouse}\!(x) \rightarrow \exists y \quad //\text{married}/\text{person}(y) \wedge //@\text{ssn}(y, x) \\
\text{(key,s,p)} & \quad \forall x, y, s \quad //\text{person}(x) \wedge //\text{person}(y) \wedge //@\text{ssn}(x, s) \wedge //@\text{ssn}(y, s) \rightarrow x = y \\
\text{(grandpa)} & \quad \forall x, y, z, u \quad //\text{(son|daughter)}(x, y) \wedge //\text{(son|daughter)}(y, z) \wedge //@\text{ssn}(x, u) \\
& \quad \rightarrow //@\text{grandparent}\!(z, u)
\end{align*}
\]

DTDs imply some of these constraints. Consider the DTD entry `<!ELEMENT person (address?, . . .)>` stating (among other things) that `person`-elements have at most one `address`-element nested within them. We express this as (oneAddress) above. With \( address^+ \) instead of `address?` we assert the existence of at least one `address`-subelement,
thus implying \((\text{someAddress})\). \((\text{idref})\) above says that the \text{spouse}-attribute of \text{person}-elements agrees with the value of the \text{ssn}-attribute of some \text{person}-element which is a child of some \text{married}-element which is a child of the root. This statement cannot be expressed as such in a DTD, which can only state (using IDREF declarations) that the \text{spouse}-attribute references some attribute of \text{unspecified} name, in an element \(y\) of unspecified tag. However, \((\text{idref})\) can be expressed in XML Schema, where “typed” IDREFS are allowed. The constraint \((\text{key}_{s,p})\) captures the fact that the \text{ssn}-attribute is a key for \text{person}-elements (this is again not DTD expressible, but can be stated in XML Schema and in [FS00, BDF+01, FL01]). \((\text{grandpa})\) expresses another useful kind of constraint which is reminiscent of relational inclusion dependencies [AHV95] but goes beyond the formalisms of [FS00, BDF+01, FL01] or XML Schema.

As we can see, inspired loosely by path specification in UNIX-like file directory systems, XPath was designed to be a (1) compact and (2) expressive notation. Its full-fledged definition has many features inspired by practical considerations. The techniques that we bring to bear in this thesis can tackle many of the features of full-fledged XPath, but not all. Our approach (explained in section 9.4) limits the XPath expressions we can analyze formally to a subclass we call \text{simple} and some significant extensions thereof, handled separately because they feature distinct complexities of the containment problem.

### 9.2 Simple XPath

Simple XPath expressions are generated by the following grammar (\(n\) is any tag or attribute name, \(v\) any variable name, and \(s\) any string constant):

\[
\text{(simple xpath) } p \ ::= \ p_1 | p_2 | /p | //p | p_1/p_2 | p_1//p_2 | p[q] | . | n | @n | @* | $v := @n |
\text{text() | id(p) | id(s)}
\]

\[
\text{(qualifier) } q \ ::= \ q_1 \text{ and } q_2 | q_1 \text{ or } q_2 | p | p = s | @n = $v | $v_1 = $v_2
\]

Simple XPath expressions feature both an extension and restrictions from the XPath standard. The extension is their ability to \text{bind variables}, needed in order to \text{express joins} on the attributes of elements reachable by distinct navigation paths. Variables are
introduced by the $ sign, and only for attribute values. The meaning of $v := @n$ is that of binding variable $v$ to the string value of the current node’s $n$-attribute. The test $@n = $v in a qualifier is satisfied if the value of the current node’s $n$-attribute equals the value $v$ was previously bound to. $v_1 = $v is satisfied if $v_1, v_2$ were bound to the same string value.

Our ability to bind variables goes beyond the XPath specification, which intends variables to be bound in the outside context (usually represented by XPointer, XSLT or XQuery expressions), and only allows for testing their values inside XPath expressions. However, in the context of using these XPaths in XQuery, this does not result in an effective increase in expressive power, since XPath expressions are not meant to be used standing alone, but rather embedded in expressions of the three standards mentioned above. In the case of XQuery for instance (which is what we ultimately want to optimize), an XPath expression that binds variables is just syntactic sugar for a query with several XPath expressions that don’t: the illegal XQuery For $x$ in /a[@n = $v]/b[@m = $v] is equivalent to the legal (even well-behaved) one For $a$ in /a, $b$ in $a/b$, Where $a/@n = b/@m$.

In fact, an inspection of the definition of well-behaved XQueries (section A.1) shows that simple XPaths can be expressed by well-behaved XQueries.

The most notable restrictions to the full-fledged standard [W3Cb] are the absence of the navigation axes following, following-sibling, preceding, preceding-sibling. This is because for the time being we disregard the document order, seeing the XML document as an unordered tree, in which these axes have no meaning 1. Some of these restrictions are lifted in section 9.5, where we handle following-sibling and preceding-sibling.

Moreover, we disallow for the moment navigation steps via the child axis from or to elements of unspecified tag. This can be done either by using the wildcard * for going to a child of unspecified tag name, or inversely, by using the parent axis to get to a parent of unspecified tag name, and finally, by using the ancestor axis, which performs an implicit parent navigation step, followed by an ancestor-or-self step. This is why *, parent, ancestor, ancestor-or-self are missing from our grammar. We treat these separately in section 9.3 because it turns out that the corresponding containment problem has higher complexity.

1Note that this view is actually consistent with the XPath 1.0 specification [W3Cb], which defines the semantics of XPath expressions as being a set of nodes. The upcoming XPath 2.0 is expected to introduce list semantics, which we do not consider here.
We also rule out negation from qualifiers, for the same reason for which negation causes problems in the classical relational dependency theory [AHV95].

**Simple XPath integrity constraints (SXICs).** We consider dependencies of the general form

\[
\forall x_1 \ldots \forall x_n [B(x_1, \ldots, x_n) \Rightarrow \bigvee_{i=1}^l \exists z_{i,1} \ldots \exists z_{i,k_i} C_i(x_1, \ldots, x_n, z_{i,1}, \ldots, z_{i,k_i})]
\]

(9.1)

where \(B, C_i\) are conjunctions of atoms of form \(v \ p \ w\) where \(p\) is a simple XPath expression or equality atoms of the form \(v = w\), where \(v, w\) are variables or constants. We demand of course that \(v, w\) be of compatible type. \(v\) may be missing from a path atom if \(p\)'s context node is the root of the document (i.e. if \(p\) begins with / or //).

All constraints shown in section 9.1 are SXICs. We have seen that some of them are not expressible by DTDs, while others are implied by them. But in general DTDs and SXICs are incomparable. DTD features that cannot be expressed by SXICs are the order of sibling elements, and the fact that an element admits subelements of given tags only.

**Satisfaction of SXICs.** We say that the binding of \(v\) to a node \(a\), and of \(w\) to a node \(b\) satisfies a path atom \(v \ p \ w\) if \(b\) is equal to some node in the set returned by \(p\) when starting from context node \(a\). We define equality as equality of the string values for text and attribute nodes, while an element node is equal only to itself. Equality atoms are satisfied according to this definition. An SXIC of general form (9.1) is satisfied if for any type-consistent binding of the variables \(x_1, \ldots, x_n\) that satisfies all atoms in \(B\), there is some \(1 \leq i \leq l\) and some extension of this binding to the variables \(z_{i,1}, \ldots, z_{i,k_i}\) such that all atoms of \(C_i\) are satisfied by the extended binding.

**Containment under SXICs.** Given a set \(C\) of SXICs, and simple XPaths \(P_1, P_2\), we say that \(P_1\) is contained in \(P_2\) under \(C\) (denoted \(P_1 \subseteq_C P_2\)) if every node in the set returned by \(P_1\) is equal (in our sense) to some node in the set returned by \(P_2\) whenever both are applied to any XML document which satisfies all SXICs in \(C\). (This definition is more flexible than just asking for containment of the node sets returned by \(P_1, P_2\), because it does not distinguish between attribute and text nodes of distinct identity but equal string value.)

**Bounded SXICs.** This subclass of SXICs allows the same generality as all SXICs in the
left-hand-side $B$ of the implication, but it restricts the form of the right-hand-side of the implication. Namely, the XPath atoms occurring in the conjunctions $C_i$ must have one of the following forms:

$$v = w \quad v ./@n w \quad v ./@* w \quad v ./n w \quad v ././. w \quad //. w \quad /n w$$

Moreover, while all occurrences of $v, w$ can be universally quantified, there are restrictions on the cases when they may be existentially quantified. In order to state these restrictions, we introduce the notion of bounded-depth variable: we say that variable $w$ is bounded-depth if it appears on either side of the implication in an atom $/n w$, or in atoms $v ./n w$ or $w ./n v$, with $v$ bounded-depth. The restrictions on existential quantification are given below:

- $w$ must be universally quantified in $v ./@n w, v ./@* w$ and $v ././. w$

- $v$ or $w$ may be existentially quantified in $v ./n w$ only if they are bounded-depth.

All SXIC examples in section 9.1 are bounded except for $(someAddress)$, which contains the existentially quantified, non-bounded-depth variable $y$.

Theorem 9.2.1 Containment of simple XPath expressions under bounded SXICs is decidable. If we fix the constraints, the problem is in $\Pi^p_2$ in the size of the expressions (if we don’t, the problem is in EXPTIME in the size of the constraints). If in addition we consider disjunction-free simple expressions and constraints, the complexity drops to NP. Moreover, if we also disallow attribute variables in the expressions, the complexity drops to PTIME.

The proof is in section D.1 of appendix D.

In practice, we often know that XML documents satisfy SXICs that are not necessarily bounded, the most salient examples being SXICs implied by DTDs, such as $(someAddress)$ from section 9.1. Unfortunately, we have the following result (see section D.1 of appendix D for proof):

Theorem 9.2.2 Containment of simple XPath expressions under unbounded SXICs is undecidable.
**Complexity lower bounds.** It turns out that for fixed SXICs, the upper bounds in theorem 9.2.1 are tight:

**Theorem 9.2.3** Just containment of simple XPath expressions (no constraints) is $\Pi_2^P$-hard. Containment of disjunction-free simple XPath expressions (again no constraints) is NP-hard.

The proof is omitted, but the result is not surprising: we show in section 9.4 how the disjunction-free simple XPath expressions are compiled to conjunctive queries against the generic relational encoding of XML, GReX, introduced in the context of compiling XQueries in section 3.1. Then simple XPaths with disjunction correspond to conjunctive queries with the union operator. While the classical NP-hardness and $\Pi_2^P$-hardness results for containment of conjunctive queries, respectively conjunctive queries with union do not imply the results on simple XPath, the latter fragment is relationally complete, so we would expect at least these lower bounds.

A missing piece in the puzzle is the lower bound for containment under simple bounded SXICs when the constraints are not fixed. We conjecture EXPTIME-hardness however, expecting that the proof of EXPTIME-hardness for the relational chase [CLM81] can be adapted.

**DTDs and SXICs.** In XML practice, constraints on the form of documents are often specified using DTDs. A natural question pertains to the status of our decidability results in the presence of DTDs, with or without SXICs. A careful analysis of the proof of theorem 9.2.2 shows the following. Let $C_1$ be a set of bounded SXICs without disjunction and existentials. Let $D$ be a DTD and let $C_2$ be a set of unbounded SXICs implied by $D$. Let also $X_1$ and $X_2$ be two simple XPath expressions. What we prove, in fact, is that the problem of whether $X_1 \subseteq C_1 \cup C_2 \ X_2$ is undecidable.

**Corollary 9.2.4** Containment of simple XPath expressions is undecidable in both following scenarios (1) under unbounded SXICs, and (2) in the presence of bounded SXICs and DTDs.

Bounded SXICs cover many common cases: given a DTD, it is usually possible to rewrite constraints such as (someAddress) in bounded syntax, unless in the rest of the DTD (which
we do not specify) the \texttt{address} element is nested (immediately or not) in some “X”-element
that may contain a descendant “X”-subelement. Such “cyclic” element declarations are
not very common!

The problem of deciding containment of simple XPath expressions under DTDs only
(no SXICs) remains open, and the following lower bound which is in fact in the size of the
expressions, combined with the upper bound in theorem 9.2.1, suggests that the techniques
that we use in this thesis are unlikely to help:

\textbf{Theorem 9.2.5} \textit{Containment of simple XPath expressions in the presence of DTDs is
\texttt{PSPACE}-hard.}

The proof is omitted. As discussed above, the lower bound carries over to well-behaved
XQueries.

\section{Beyond Simple XPaths}

In this section we enrich simple XPath expressions with several navigation primitives from
the XPath standard.

\textbf{Parent axis.} We allow navigation to the parent of the current node. Concretely, this
amounts to adding the production \( p ::= \texttt{parent} \) to the grammar in section 9.2.

\textbf{Ancestor axes.} We allow navigation along the ancestor and ancestor-or-self axis. The
corresponding productions are \( p ::= \texttt{ancestor} | \texttt{ancestor-or-self} \).

\textbf{Wildcard Child.} We further allow navigation along the child axis to elements of
unspecified tag, adding \( p ::= * \) to our grammar (\(* \) is called the \textit{wildcard}). Here is an XPath
expression using wildcard child navigation:

\[ P' \overset{\text{def}}{=} //c/*[@m = "0"] \, . \]

It returns the set of elements of unspecified tag (indicated by the \(* \)), that have an
\( m \)-attribute of value "0" and a parent tagged \( c \) who is a non-immediate descendant of the
document root. Note the use of the wildcard \(* \) (disallowed in simple XPath expressions).

\textbf{Path equality.} We extend the grammar of qualifiers with the production \( q ::= p_1 = p_2 \),
corresponding to path equality tests. Such tests are satisfied if some node returned by path
\( p_1 \) is equal to some node returned by path \( p_2 \). Equality tests must of course typecheck, and
they are satisfied for text and attribute nodes if and only if the string values are equal. In contrast, an element node is only equal to itself. This definition of element node equality follows XML-QL [DFH99] as opposed to the ad-hoc treatment in [W3C].

Although none of the above extensions seems to have anything to do with disjunction, each one of them (except parent, for which we do not know what happens) when added to the disjunction-free fragment raises the complexity (recall from theorem 9.2.1 that it is NP when the constraints are fixed):

**Theorem 9.3.1** Adding any one of the following to disjunction-free simple XPath expressions makes their containment problem (no constraints) \( \Pi_2^p \)-hard: 1. path equality 2. ancestor axis 3. ancestor-or-self axis 4. wildcard child

However, this is pretty much as far as the complexity raises (see section D.2 in appendix D for a proof theorem 9.3.1 part (4). The proof of theorem 9.3.2 is an easy application of theorem D.1.1 from section D.1.1):

**Theorem 9.3.2** Consider simple XPath expressions enriched with path equality and ancestor-or-self axis. The containment of such expressions under fixed bounded SXICs is in \( \Pi_2^p \) in the expression size.

In dealing with wildcard we have further restricted the constraints. We believe however that this restriction can be lifted.

**Tree SXICs.** These are bounded SXICs that satisfy the following additional restrictions: (i) \( v \) must be universally quantified in \( v /n w \) (recall page 92). (ii) We disallow atoms of form \( v ./ w \) from the right-hand side of the implication and (iii) For any constraint \( c \) and any of \( c \)'s equality atoms of form \( v = w \) (where \( v, w \) are variables) in the right-hand side of the implication, if \( v, w \) are bound to element nodes, \( c \) must contain the atoms \( u /n v \) and \( u /m w \) for some variable \( u \) and tag names \( n, m \).

Restriction (iii) ensures that in all models satisfying \( c \), the only expressible key constraints are keys among sibling nodes. Recalling the examples in section 9.1, (oneAddress) corresponds to this restriction, while (key,s,p) does not. The intuition behind all three restrictions is that no combination of tree SXICs can compromise the tree property of a given document, whence their name.
Theorem 9.3.3: Consider simple XPath expressions enriched with parent, ancestor, ancestor-or-self and wildcard child navigation. The containment of such expressions under fixed tree SXICs is in \( \Pi_2^p \) in the expression size.

The proof is in section D.2.

9.4 Decision Technique: First-Order Translation

We use the same decision technique as for containment of XQueries: reducing containment under constraints to an equivalent first-order question. Recall that in section 3.1, we defined TIX which consists of a relational schema and some first-order integrity constraints on this schema. We translate XPath expressions into unions of relational conjunctive queries over the schema of TIX. Moreover, we translate SXICs into first-order sentences over the same schema and of the same form as the integrity constraints in TIX. Denoting the translation of a set of SXICs with \( C \), we reduce containment of simple XPath expressions under \( C \) to containment of unions of relational conjunctive queries under TIX \( \cup \Sigma_C \).

Section D.1 shows in detail how we use this reduction to containment of relational queries under first-order dependencies to decide containment of simple XPath expressions, and section D.2 shows how we extend this kind of reasoning to handle containment for the extensions of simple XPath mentioned in section 9.3.

Since the simple XPath fragment enriched with path equality and the two versions of ancestor navigation, is contained in that of well-behaved XQueries, the decision procedure differs from theorem C.3.1 only in the analysis of the termination of the chase with TIX \( \cup \Sigma_C \), which follows from the boundedness of the SXICs.

Why Wildcard Child Navigation is Not Well-Behaved: However, the enrichment with wildcard child navigation is not covered by our treatment of well-behaved XQueries, and for a good reason: it turns out that the dependencies in TIX become insufficient in reasoning about wildcard expressions. See example D.2.1 in section D.2 for two XPaths containing wildcard, such that their containment holds, but the containment of their relational translations does not hold over TIX-instances. This is the reason why our completeness result for XQuery reformulation (theorem 6.2.1 in section 6.2) does not hold for
ill-behaved XQueries.

9.5 Extension to Order

**Order.** Our decision procedure for containment extends straightforwardly if we add the `preceding-sibling` and `following-sibling` navigation steps to the fragments of XPath we show in section 9.3, and the complexity results carry over to this extension. We consider the ordered XML data model, and extend our XPath fragments with order-related predicates:

\[
p ::= \text{preceding-sibling}(p) | \text{following-sibling}(p) | \text{preceding}(p) | \text{following}(p)
\]

We take the view of the XPath 1.0 specification [W3Cb], according to which an XPath expression evaluates to a node set, thus restricting the impact of order only to checking the predicates above (this situation will change with the XPath 2.0 specification however).

Let \(TIX^O\) be the extension of TIX with the binary relations `preceding-sibling`, `preceding` and the constraints

\[
\begin{align*}
\text{(trans}_{ps}\text{)} & \quad \forall x, y, z \ [\ \text{preceding-sibling}(x, y) \land \text{preceding-sibling}(y, z) \rightarrow \text{preceding-sibling}(x, z)] \\
\text{(min}_{ps}\text{)} & \quad \forall x, y \ [\ \text{preceding-sibling}(x, y) \rightarrow \exists z \ \text{child}(z, x) \land \text{child}(z, y)] \\
\text{(total}_{ps}\text{)} & \quad \forall x, y, z \ [\ \text{child}(x, y) \land \text{child}(x, z) \rightarrow y = z \lor \text{preceding-sibling}(y, z) \lor \text{preceding-sibling}(z, y)] \\
\text{(base}_{p}\text{)} & \quad \forall x, y, z, u \ [\ \text{desc}(x, z) \land \text{preceding-sibling}(x, y) \land \text{desc}(u, y) \rightarrow \text{preceding}(y, u)] \\
\text{(trans}_{p}\text{)} & \quad \forall x, y, z \ [\ \text{preceding}(x, y) \land \text{preceding}(y, z) \rightarrow \text{preceding}(x, z)] \\
\text{(min}_{p}\text{)} & \quad \forall x, y \ [\ \text{preceding}(x, y) \rightarrow \exists u, v \ \text{preceding-sibling}(u, v) \land \text{desc}(u, x) \land \text{desc}(v, y)] \\
\text{(total}_{p}\text{)} & \quad \forall x, y \ [\ N(x) \land N(y) \rightarrow \text{desc}(x, y) \lor \text{preceding}(x, y) \lor \text{preceding}(y, x)]
\end{align*}
\]

We provide the first-order translation

\[
\begin{align*}
T(x, \text{preceding-sibling}(p), y) & = \{\text{preceding-sibling}(x, y)\} \cup T(y, p, z) \\
T(x, \text{preceding}(p), y) & = \{\text{preceding}(x, y)\} \cup T(y, p, z) \\
T(x, \text{following-sibling}(p), y) & = T(y, \text{preceding-sibling}(p), x) \\
T(x, \text{following}(p), y) & = T(y, \text{preceding}(p), x)
\end{align*}
\]
Theorem 9.5.1 If we add the preceding-sibling and following-sibling predicates to the XPath fragments in section 9.3 and use $TIX^O$ above, the algorithm in section D.2 remains a decision procedure for containment, and the complexity results carry over.

If the XPath expressions contain following and preceding as well, the algorithm remains sound, but we do not know if it is complete for deciding containment.

**What we do not capture.** The order-related features we do not capture in this way are index and range qualifiers. The expression `/a[2]` uses the index qualifier 2 to return the second $a$-child of the root. `/a[range 2 to 4]` returns the second, third and fourth $a$-child.
XQueries have list semantics. In this thesis, we ignore (most aspects of) order, thus solving the problem of reformulation for XQueries with bag semantics (such XQueries can be explicitly written, using the keyword unordered). It turns out that in most cases, the bag semantics coincides with set semantics, or can be reduced to it by making use of the identities of nodes in the XML tree. There is one exception however, namely the concatenation operator. This operator may introduce duplicate occurrences of the same node identity into the list of variable bindings of an XQuery.

While our motivation is the reformulation of XQueries with bag semantics, we start by considering SQL queries. The extension of our results to XQuery is the object of future research.

We have previously proposed chase and backchase as a novel method for using materialized views and integrity constraints in query optimization. In this chapter, we show that the method is usable in realistic optimizers by extending it to bag and mixed (i.e. bag-set) semantics as well as to grouping views and by showing how to integrate it with standard cost-based optimization. We understand materialized views broadly, including user-defined views, cached queries and physical access structures (such as join indexes, access support relations, and gmaps). Moreover, our internal query representation supports object features hence the method applies to OQL and (extended) SQL:1999 queries.
Chase and backchase supports a very general class of integrity constraints, thus being able to find execution plans using views that do not fall in the scope of other methods. In fact, we prove completeness theorems that show that our method will find the best plan in the presence of common and practically important classes of constraints and views, even when bag and set semantics are mixed (as in rewriting a bag query using a mix of set and bag views).

10.1 Introduction

Optimizers are characterized by three dimensions: the search space for execution plans, the strategy for exploring it, and the cost model used to compare plans to each other. This work addresses the first two dimensions.

Rather than comparing optimizers by exhibiting single examples of plans that one may find while another may not, it would be desirable to have mathematical results guaranteeing that at least under certain conditions the optimal plan is not missed (so-called completeness results). Such results are only possible if there is a clear definition of the search space for alternate plans, which is conceptually independent of the exploration strategy. Even when completeness can only be shown under restrictions, a clearly defined search space improves the overall understanding of the tradeoff between plan quality and optimization speed.

For early optimizers, the search space was well-understood: consider all available access methods (file scan or index lookups) per relation, all possible (usually only “left-deep”) join orderings, and all join algorithms (from a small set of well-established ones). Exploration strategies included exhaustive search enhanced with cost-based pruning (e.g. dynamic programming [SAC+79]), and forms of non-exhaustive search trading plan quality (i.e. closeness to optimality) for optimization speed-up: rule-based [GCD+94], heuristic, randomized [IW87].

Subsequent optimizers extended the search space in two different directions, in order to improve plan quality, but also to increase physical data independence. One direction was semantic optimization, that is rewriting a query according to integrity constraints
in the logical schema\(^1\). The other consisted in rewriting the query to also use, e.g., materialized views/cached queries [ACPS96, Hal01, BDD\(^+\)98, ZCL\(^+\)00] or physical storage structures [Val87, KM90, TSI94]. In both cases, the rewriting was combined with standard cost-based pruning.

Completeness results were provided in the absence of integrity constraints [CKPS95, LMSS95]. However, previous attempts to combine the use of views and integrity constraints all have the same problem: an ad-hoc search space for rewritings, usually defined implicitly by the exploration strategy, in terms of a particular set of transformations that are attempted. This makes it hard to find conditions that guarantee the completeness of the optimizer.

In a VLDB’99 paper [DPT99] we have introduced the \textit{chase & backchase} method in answer to some of these problems. Since then we have made significant progress in making this approach practical. We report this progress here. The following motivating example will help us summarize some of our new contributions.

\subsection{Motivating Example}

Consider the following ODMG/ODL schema definition, in which \texttt{Teams} associates projects with names of their members, \texttt{Payroll} associates departments with employees and \texttt{Dept} is a class whose set-valued attribute \texttt{DProjs} contains the projects run by each department. We will need to consider both \texttt{set} and \texttt{bag} semantics hence the \texttt{Set/Bag}, and we use a \texttt{b}

\footnote{There is a large body of work on this topic. Since we have reviewed it in [DPT99] and given the space constraints of this submission, we will omit here those references.}
superscript for the bag semantics: Teams\textsuperscript{b}, Payroll\textsuperscript{b}.

```
class Dept

(extent depts key DName){
    attribute string DName;
    attribute set<string> DProjs;
}
```

Teams/Teams\textsuperscript{b}: set/bag<struct{
    string TProj;
    string TMember;
}> Payroll/Payroll\textsuperscript{b}: set/bag<struct{
    string PDept;
    string Empl;
}>

Consider also an ODMG/OQL query \( Q \) that returns the members of projects run by the "Security" department. Here too we consider set and bag semantics: \( Q \) is \textit{with distinct}, while \( Q^b \) is without.

\[
Q/Q^b : \text{select } \text{distinct } \text{struct}(E : t.TMember) \\
\quad \text{from } depts d, d.DProjs pn, Teams t \\
\quad \text{where } pn = t.TProj \text{ and } d.DName = "Security"
\]

Assume that all employees must receive reports about all projects run by their department. This is facilitated by materializing the view \( V_1 \) below. Assume also the materialized view \( V_2 \) below that joins Teams and Payroll. We consider both set and bag semantics for these views. Note also that the schema does not require that project members also be employees.

\[
V_1/V_1^b : \text{select } \text{distinct } \text{struct}(D : d.DName, \\
\quad P : pn, E : p.Empl) \\
\quad \text{from } depts d, d.DProjs pn, Payroll p \\
\quad \text{where } d.DName = p.PDept
\]
We will use this schema, query and views to illustrate two important aspects that have guided us:

1. **Integrity constraints can help in answering queries using views.** A query may not in general be equivalent to another query that uses given views. But additional equivalences may hold in the presence of integrity constraints. Hence an equivalent query that uses the views might be found. We explain this aspect in set semantics. For example, $V_1, V_2$ cannot in general be used to answer $Q$: if there are some team members of projects run by "Security" that are not on the Payroll (outside consultants), then any combination of $V_1$ and $V_2$ will miss these, because both views only store persons on the Payroll.

   The situation changes if we assume the integrity constraint (insider) that states that "Security" uses only its own employees on the projects it runs.

   $\forall (d \in \text{depts})(p \in d.\text{DProjs})(t \in \text{Teams}) pn = t.\text{TProj} \land d.\text{DName} = \text{"Security"}$

$\rightarrow \exists (p \in \text{Payroll}) p.\text{PDept} = d.\text{DName} \land p.\text{Empl} = t.\text{TMember}$

Then, $Q$ is equivalent to $R$:

$$R/R^b : \text{select [distinct] struct(E : v_1.E)}$$

$$\text{from } V_1 v_1, V_2 v_2$$

$$\text{where } v_1.\text{D} = \text{"Security"}$$

and $v_1.\text{P} = v_2.\text{P}$ and $v_1.\text{E} = v_2.\text{E}$

and $v_1.\text{D} = v_2.\text{D}$
Our approach is the first that considers answering queries using views in the presence of integrity constraints general enough to include for example (insider). In particular the methods in [CKPS95, LMSS95, GHQ95, GBLP96, SDJL96, BDD+98, ZCL+00] will not discover the rewriting of $Q$ to $R$.

2. When the schema, query, or views have bag semantics, we may need stronger integrity constraints. This is because the multiplicities (number of copies) of tuples may be different depending on whether the views are used or not. Stronger integrity constraints can insure this does not happen. For instance, if $Payroll^b$ contains the duplicate 2 entry ($PDept: \text{"Security"}, Empl: \text{"John"}$), and "John" is a member of project "p1" run in the "Security" department, $V_1^b$ will return two tuples ($D: \text{"Security"}, P: \text{"p1"}, E: \text{"John"}$) and so will $V_2^b$. Hence, $R^b$ will return four tuples ($E: \text{"John"}$), as opposed to only one returned by $Q^b$. The integrity constraint (insider) is not strong enough to prevent this. What we would like to say in this constraint is not just that ... $\exists (p \in Payroll)$ ... but that there exists a unique such $p$! A possible notation for this would be:

\[
\text{(insider}^b) \quad \forall (d \in \text{depts})(pn \in d.\text{Projs})(t \in \text{Teams}^b)pn = t.\text{TProj} \land d.\text{Name} = \text{"Security"} \\
\quad \rightarrow \exists ! (p \in Payroll^b)p.\text{PDept} = d.\text{Name} \land p.\text{Empl} = t.\text{TMember}
\]

However, we have to be careful because $\text{Teams}^b$ and $\text{Payroll}^b$ are bags, not sets, and therefore standard logic does not give a meaning to such an assertion. This leads us to consider in this chapter a new class of integrity constraints for bag semantics that we call Unique Witness Dependencies (UWDs). Their definition and the exact meaning of (insider$^b$) is given in section 10.2.

It turns out that the common practice of asserting key and referential integrity (foreign key) constraints in bag semantics schemas corresponds to a UWD. Our method handles answering queries using views in the presence of such constraints and finding query rewritings that previous approaches miss, for example rewriting $Q^b$ to $R^b$ above in the presence of (insider$^b$) (more examples in section 10.2).

---

2Schema relations are often duplicate-free. But the problem really stems from the bag semantics of the views. Assume that $\text{Payroll}^b$ has additional attributes (e.g., $\text{ContractNo}$). Even if $\text{Payroll}^b$ is duplicate-free, if its projection on $\text{PDept}$ and $\text{Empl}$ has duplicates then we have the same multiplicity problem.
10.1.2 Contributions and Some Related Work

**Mixed semantics** In practice, schema elements are often sets, while views and queries are often bags, defined without using the `distinct` keyword (bag-set semantics). We develop here techniques for bag semantics, bag-specific constraints (UWDs), and for handling bag queries over arbitrary mixes of bag and set schema elements and views (this includes bag-set semantics), building on our previous results on set semantics [DPT99].

The main novelty of the C&B method (we use C&B as a shorthand for chase & backchase) is in the way it constructs the search space for rewritings, namely as the result of chasing the query with constraints that capture materialized views as well as integrity constraints [DPT99]. We understand materialized views broadly, including user-defined views, cached queries and physical access structures (such as join indexes [Val87], access support relations [KM90], and gmaps [TSI94]). The chase result, a query we called the “universal plan”, gathers redundantly all the ways to answer the original query. The search space is defined by the subqueries of the universal plan and explored by backchase. Thus we do rewriting with views, semantic optimization, and query minimization all in one.

**Completeness** The theoretical contributions of this chapter are theorems that state that the C&B method is *complete*, i.e., will always find an optimal plan if one exists, using given views and under given constraints. A preliminary result in [DPT99] showed completeness only in the absence of integrity constraints and only for set semantics. Here we prove completeness for mixed semantics and both set and bag integrity constraints.

Of course, such theorems are limited by standard undecidability and incompleteness barriers. Still, ours appear to be the first completeness results that assume very general and practically relevant classes of integrity constraints. Algorithms that are complete in the absence of constraints are given in [CKPS95] for bag semantics and in [LMSS95] for set semantics. The algorithm in [DG97] is complete in the presence of just functional dependencies.

This chapter continues [PT99, DPT99, PDST00, Pop00] in using for queries, views and plans a language which uses *dictionaries* to express object-relational and object-oriented
features, as in ODMG and in SQL:1999 and its extensions, as well as a variety of physical access structures, including indexes. The integrity constraints are expressed in a logic that corresponds to the same language, capturing common OO/relational integrity constraints, such as functional dependencies/key constraints, referential integrity (foreign key), lossless join and inverse relationship constraints. The C&B algorithm is defined on the path-conjunctive restriction of this language, while the completeness theorems require constraints to be full (so that the chase is guaranteed to terminate). To avoid cluttering the early sections, we postpone the precise definitions of the plan and constraint language to section 10.7.

The structure of the optimizer we proposed in [DPT99] separated C&B optimization as a stage preceding standard cost-based query optimization. In fact, it was not clear that the chase or backchase were feasible at all, given the theoretical intractability of the chase [AHV95]. Subsequently, we developed implementation techniques for the chase and the backchase and reported on their feasibility in [PDST00]. That work still kept the C&B stage separate from any cost-based processing. However, there were strong indications that the C&B search space exploration could itself exploit cost information.

**Backchase with cost-based pruning** The main practical contribution of this chapter is that we show how to restructure the exploration of the search space to allow for cost-based pruning using dynamic programming and we report on experimental results showing the benefit of combining our technique with System R-style cost-based optimization. As an additional improvement we show that we can postpone primary and secondary index selection to the cost-based phase, reducing the number of constraints used in chasing while preserving completeness.

The complete (no constraints, pure bag semantics) algorithm in [CKPS95] includes cost-based dynamic programming and, in fact, in the absence of integrity constraints our algorithm does the same work (same efficiency and same simplicity in implementation). Minicon [PL] builds on the ideas of [LMSS95] and improves scalability with the number of views, however the algorithm is designed for a different purpose (data integration) and it doesn’t consider any integrity constraints. The algorithm in [TSI94], is also integrated with cost-based dynamic programming but uses a restricted class of constraints, is restricted to
SPJ queries and views with set semantics in which every relation appears at most once, and is incomplete \textsuperscript{3}.

**Grouping** In this chapter we extend our theoretical and practical work to cover a class of grouping views, with both set and bag semantics and with the same classes of set and bag constraints. For the completeness theorems we still have an important restriction: the grouping views are indeed nested (at arbitrary depth in fact) but the queries themselves have flat output. Also, we can decide equivalence of grouping queries in the presence of the same integrity constraints.

Even though we do not know if the method is complete, we can still apply the C&B to a nested query by rewriting each block. Then, the method can be combined with some of the techniques for handling aggregates in [SDJL96, BDD+98, ZCL+00] and thus be applied to certain queries and views with aggregation. Space does not allow us to get into details here.

Algorithms that perform query rewriting with views in Oracle and DB2 are described in [BDD+98] (with some aggregation and some classes of constraints) and in [ZCL+00] (concentrates on aggregation). A decision procedure for so-called weak equivalence of nested set queries but in the absence of integrity constraints is given in [LS97]. Both [SDJL96, CNS99] rewrite aggregate queries using aggregate views, but no integrity constraints are exploited and no queries with nested output are considered. [SDJL96] also gives a completeness result but for views without grouping and aggregation.

**Overview of the rest of the chapter.** In section 10.2 we define and justify the (new) class of bag constraints that we can exploit systematically in C&B. In section 10.3 we review (following [DPT99]) the C&B method for just set semantics. In section 10.4 we discuss the (new) chase and backchase of bag queries with bag integrity constraints and bag views. The (new) completeness theorem for quite general classes of queries, views and constraints, and with both bag and set semantics, is in section 10.5. The (new) algorithm that performs the backchase phase in a bottom-up fashion in conjunction with dynamic-programming cost-based pruning is described in section 10.6. Section 10.7 gives formal details on the query and constraint languages and the chase in bag semantics.

\textsuperscript{3}Purposefully so, since they propose a PTIME algorithm for an NP-hard problem.
In section 10.8 we sketch our (new) extension of the C&B method to grouping views. Conclusions and further work complete the chapter.

10.2 Integrity Constraints in Bag Semantics

Previous work on rewriting using views in bag semantics does not give any systematic way of exploiting integrity constraints. However, SQL schemas do assert some constraints and these sometimes enable the use of views. For instance, revisiting the motivating example in section 10.1.1, the bag query $Q^b$ can be rewritten as:

$$
R^b_1: \text{select } \text{struct}(E: v_2.E) \\
\text{from } \text{depts } d, d.DProjs pn, V_2 v_2 \\
\text{where } d.DName = "Security" \text{ and } pn = v_2.P \text{ and } d.DName = v_2.D
$$

provided that $\text{Payroll}^b$ contains no duplicates, $\text{Empl}$ is a key in $\text{Payroll}^b$, and $\text{TMember}$ is a foreign key in $\text{Teams}^b$ referencing $\text{Empl}$ in $\text{Payroll}^b$. In fact, it suffices for $\text{Payroll}^b$ to not contain duplicates of just the tuples mentioning project team members (in addition to the key and foreign key constraints).

Of course, SQL itself does not provide a general notation for stating assertions about bags. However, the C&B method relies on such a notation. For a bag $M$, let us denote by $\text{dom } M$ the set of values of its elements and by $M[m]$ the number of occurrences of $m$ ($m$’s multiplicity in $M$). Now we can state the key \(^4\) and the foreign key constraints as

$$
(fk) \quad \forall (t \in \text{dom Teams}^b) \exists (p \in \text{dom Payroll}^b). t.TMember = p.Empl \\
(key) \quad \forall (p_1 \in \text{dom Payroll}^b)(p_2 \in \text{dom Payroll}^b). p_1.Empl = p_2.Empl \rightarrow p_1 = p_2
$$

and the constraints about no duplicates in $\text{Payroll}^b$ and in just the tuples mentioning project team members as

$$
(noDupPayroll) \quad \forall (p \in \text{dom Payroll}^b). \text{Payroll}^b[p] = 1 \\
(noDupPayrollTeams) \quad \forall (t \in \text{dom Teams}^b). (p \in \text{dom Payroll}^b). t.TMember = p.Empl \rightarrow \text{Payroll}^b[p] = 1
$$

\(^4\)For us a key constraint is just a functional dependency and it does not imply duplicate-freeness as opposed to the UNIQUE constraint in SQL, which is actually the combination of our (key) and (noDupPayroll) below.
Note that \((\text{noDupPayroll})\) implies \((\text{noDupPayrollTeams})\) and that \((\text{key}) + (fk) + (\text{noDupPayrollTeams})\) suffice to justify rewriting \(Q^b\) as \(R^b_1\). A similar combination of three assertions captures what was intended by \((\text{insider}^b)\) in section 10.1.1 to justify \(R^b\). Our analysis of constraints in bag semantics reveals this pattern of three assertions as occurring often. We therefore introduce a name and a notation which bundles together such a combination of assertions. Section 10.4 will show that this choice is not arbitrary: it turns out that these allow sound rewriting of queries with bag semantics in an analogous way to set semantics.

**Definition 10.2.1 (Unique Witness Dependencies (UWDs))** Given an instance \(I\) in which \(M, N\) are bags, we say that the dependency \(d\) denoted

\[
(d) \forall (m \in M) \; B_1(m) \rightarrow \exists (n \in N) \; B_2(m, n) \quad (M, N \text{ are bags!})
\]

is satisfied by \(I\) if and only if the following are true in \(I\):

- **(witness)** \(\forall (m \in \text{dom} M) \; [B_1(m) \rightarrow \exists (n \in \text{dom} N) \; B_2(m, n)]\)
- **(functionality)** \(\forall (m \in \text{dom} M) \; (n \in \text{dom} N) \; (n' \in \text{dom} N) \; [B_1(m) \land B_2(m, n) \land B_2(m, n') \rightarrow n = n']\)
- **(multiplicity)** \(\forall (m \in \text{dom} M) \; (n \in \text{dom} N) \; [B_1(m) \land B_2(m, n) \rightarrow N[n] = 1]\)

\(B_1, B_2\) are conjunctions of equalities of variables or their projections on attributes \(^5\). The existentially quantified variable may be absent, in which case the meaning of \(\forall (m \in M) B_1 \rightarrow B_2\) is the simplification of \((\text{witness})\) above.

Observe that \((fk), (key), (\text{noDupPayrollTeams}),\) are respectively the \((\text{witness}), (\text{functionality}), (\text{multiplicity})\) of the UWD

\[
(fk^b) : \forall (t \in \text{Teams}^b) \exists! (p \in \text{Payroll}^b) \; t.\text{Member} = p.\text{Empl}
\]

UWDs also allow us to handle applications mixing bags and sets, by considering all database relations as bags by default, and stating for some of them that they allow no duplicates. For \(\text{Payroll}^b\), this is expressed by the UWD \(\forall (p \in \text{Payroll}^b) \exists! (p' \in \text{Payroll}^b) p' = p.\)

\(^5\)We actually allow equalities of path expressions, which generalize projections and are detailed in section 10.7. Also, we omit the straightforward generalization to multiple quantified variables.
Indeed, the corresponding (witness) and (functionality) dependencies are trivially satisfied, while the (multiplicity) dependency is trivially equivalent to \((\text{noDup}_{\text{Payroll}})\) above. As a particular case, we use this kind of “no-duplicates” constraints to model so-called bag-set semantics (bag queries over set relations [CV93]).

We show in section 10.4 how we use UWDs to rewrite bag queries. Our approach automatically detects and exploits all UWDs implied by common integrity constraints (like \((\text{fk}^b)\)). Because of space limitations, we do not elaborate on this point.

**Remark.** The fact that \((\text{insider})\) enables the rewriting \(R\) and \((\text{insider}^b)\) enables \(R^b\) does not mean that in general we can reduce the problem of rewriting with bag semantics to set semantics. To see this, construct \(V'_2\) from \(V_2\) by dropping the \(D\) attribute from its output, and \(R'\) from \(R\) by using \(V'_2\) instead of \(V_2\) and dropping \(v_1.D = v_2.D\) from \(R\)'s where clause. Now \((\text{insider})\) enables the rewriting \(R'\), but \((\text{insider}^b)\) does not enable \(R^b\), even if \(\text{Payroll}^b\) contains no duplicates! This is because an employee may be on the payroll of several departments, again affecting the multiplicities of some tuples in \(R^b\)'s answer: suppose that \(\text{Payroll}\) contains two entries for John, \((\text{PDept} : "Retail", \text{Empl} : "John")\) and \((\text{PDept} : "Security", \text{Empl} : "John")\). Then \(V^b_1\) returns \((D : "Security", P : "p1", E : "John")\), and \(V^b_2\) returns both this tuple and \((D : "Retail", P : "p1", E : "John")\). \(R^b\) hence returns \((E : "John")\) twice, as opposed to only once by \(Q^b\). We must have an additional “bag key constraint” for \(\text{Payroll}^b\) on its \(\text{Empl}\) attribute to infer that \(R^b\) is an equivalent rewriting. This is expressed as the UWD \((bk) \forall (p_1 \in \text{Payroll}^b) (p_2 \in \text{Payroll}^b) p_1.\text{Empl} = p_2.\text{Empl} \rightarrow p_1 = p_2\), and our algorithm obtains \(R^b\) in its presence.

### 10.3 Review of the C&B Algorithm for Set Semantics

We are given a set \(V\) of views (in the broad sense), each individual view \(V_i\) from \(V\) being defined by a query \(QV_i\). In the C&B approach, each \(V_i\) is captured with a pair of inclusion constraints: \(V_i \subseteq QV_i\) and \(QV_i \subseteq V_i\). Denote the set of all such pairs of constraints with \(\Sigma_V\). Then a rewriting of a query \(Q\) against the logical schema \(S\) in the presence of the integrity constraints in \(\Sigma_C\) is found as a query written in terms of the views in \(V\), equivalent to \(Q\) on every \(S \cup V\)-instance satisfying \(\Sigma_C \cup \Sigma_V\).

**Example.** Recalling the motivating example from section 10.1.1, the constraints capturing
\[ V_1 \text{ are} \]
\[(c_{V_1}) \quad \forall (d \in \text{depts})(p_n \in d.DProjs)(p \in \text{Payroll}) \]
\[d.DName = p.PDept \rightarrow \exists (v_1 \in V_1) v_1.D = d.DName \land v_1.P = p_n \land v_1.E = p.\text{Empl}]\]
\[(b_{V_1}) \quad \forall (v_1 \in V_1) \exists (d \in \text{depts})(p_n \in d.DProjs)(p \in \text{Payroll})\]
\[d.DName = p.PDept \land v_1.D = d.DName \land v_1.P = p_n \land v_1.E = p.\text{Empl}\]

For reasons that will become apparent shortly, \((c_{V_1})\) is called \(V_1\)'s chase-in dependency, and \((b_{V_1})\) is called its backchase dependency.

The algorithm has two phases: the first is called the chase, and it rewrites \(Q\) with the semantic constraints in \(\Sigma_C\) and the chase-in dependencies in \(\Sigma_V\), obtaining a query called the universal plan \(UP\), which explicitly mentions all physical access paths that can be used to answer \(Q\). The second phase is the backchase, which searches for \(P\) among the subqueries (defined shortly) of \(UP\).

**Phase 1: Chase.** The constraints we use in rewriting belong to the class of Embedded Path-Conjunctive Dependencies (EPCDs), which is defined in section 10.7 and only illustrated through examples here. EPCDs are logical assertions of the form \(\forall (s_1 \in S_1) \ldots \forall (s_k \in S_k)[B_1 \rightarrow \exists (t_1 \in T_1) \ldots \exists (t_l \in T_l) B_2]\). The corresponding chase step (in its simplest form) with an EPCD \(d\) is the rewrite

\[
\text{select} \quad \text{distinct} O(s_1, \ldots, s_k) \quad \text{select} \quad \text{distinct} O(s_1, \ldots, s_k) \\
\text{from} \quad \ldots, S_1 s_1, \ldots, S_k s_k, \ldots \quad \text{from} \quad \ldots, S_1 s_1, \ldots, S_k s_k, T_1 t_1, \ldots, T_l t_l, \ldots \\
\text{where} \quad \ldots \text{and} B_1 \text{ and} \ldots \quad \text{where} \quad \ldots \text{and} B_1 \text{ and} B_2 \text{ and} \ldots 
\]

**Example.** Our example query \(Q\) chases in one step using the EPCD (insider) to the following query (notice how the new variable binding \text{Payroll} \(p\) and the conditions involving \(p\) are added to \(Q\)):

\[Q_1 : \quad \text{select} \quad \text{distinct} \text{struct}(E : t.TMember) \]
\[\text{from} \quad \text{depts} \, d, \, d.DProjs \, p_n, \, \text{Teams} \, t, \, \text{Payroll} \, p \]
\[\text{where} \quad p_n = t.TProj \text{ and} \, d.DName = \text{"Security"} \quad \text{and} \, p.PDept = d.DName \text{ and} \, p.\text{Empl} = t.TMember \]

The chase phase consists in repeatedly performing chase steps with any applicable constraint from \(\Sigma_C \cup \Sigma_V\). “Applicability” must be defined carefully to avoid unnecessary
duplication of variable bindings and to allow for chasing even when query and constraint
do not match syntactically as in the example, but are related by the existence of a homomorphism from \( s_1, \ldots, s_k \) to the variables of the query. This notion is defined in section 10.7 as introduced in [PT99], which shows that the resulting chase procedure is a generalization of the classical relational chase [AHV95] to our richer query language and constraints. [PT99] also shows that, while in general the chase may not terminate, it does so for the class of full EPCDs (see section 10.7), yielding a unique result \( UP \) whose size is polynomial in that of \( Q \).

**Example.** \((\text{insider})\) is a full EPCD. By bringing the variable binding and associated conditions for \( p \) into \( Q_1 \), the chase step with \((\text{insider})\) enables further chase steps of \( Q_1 \) with \( c_{V_1} \) and \( c_{V_2} \) to obtain \( UP \) (no other chase steps apply).

\[
UP : \quad \text{select} \quad \text{distinct struct}(E : t.	ext{TMember})
\]

\[\text{from} \quad \text{depts} \; d, \; d.	ext{DProjs} \; pn, \; \text{Teams} \; t, \; \text{Payroll} \; p, \; V_1, \; V_2, \; v_1, \; v_2
\]

\[\text{where} \quad pn = t.	ext{TProj} \; \text{and} \; d.	ext{DName} = "\text{Security}" \; \text{and} \; p.	ext{PDept} = d.	ext{DName} \; \text{and} \; p.	ext{Empl} = t.	ext{TMember}
\]

\[\text{and} \quad v_1.D = d.	ext{DName} \; \text{and} \; v_1.P = pn \; \text{and} \; v_1.E = p.	ext{Empl}
\]

\[\text{and} \quad v_2.D = p.	ext{PDept} \; \text{and} \; v_2.E = t.	ext{TMember} \; \text{and} \; v_2.P = t.	ext{TProj}
\]

Notice that the effect of chasing with \( c_{V_1} \) and \( c_{V_2} \) is to explicitly bring into \( UP \) the views \( V_1, V_2 \), as well as the join conditions that relate them to the rest of the query. This wouldn’t be possible in the absence of \((\text{insider})\), and rightfully so, because \( V_1, V_2 \) wouldn’t be usable for answering \( Q \). ●

**Phase 2: Backchase.** Conceptually, the backchase phase considers all subqueries of the universal plan and keeps those which are equivalent to \( Q \) and for whom the removal of any variable binding (called scan) would compromise this equivalence. We call such queries scan-minimal rewritings of \( Q \) in the presence of \( \Sigma_C \). They are important because under a monotonic cost assumption (i.e. when the execution cost of a query is always greater than the cost of any of its subqueries), a scan-minimal query is always cheaper than its non-minimal superqueries \(^6\).

A subquery \( SQ \) is obtained by picking a subset \( SV \) of \( UP \)'s variable bindings, as well as

\(^6\)There is also a technical reason: there are infinitely many non-minimal rewritings (just duplicate some scan repeatedly), hence no hope for completeness for them. In focussing on scan-minimal rewritings, we follow [LMSS95].
all equalities implied by the UP’s where clause and involving solely the picked variables\textsuperscript{7}. The subquery’s select clause must depend only on its variables, and this is possible only if UP’s conditions imply the equality of its own select clause with that of SQ. In this case we say that SV induces SQ.

**Example.** $R$ from section 10.1.1 is a subquery of UP induced by $V_1 v_1, V_2 v_2$ since for instance $v_1.D = v_2.D$ is implied by $v_1.D = d.DName, d.DName = p.PDept$ and $p.PDept = v_2.D$. $R_1$ from section 10.2 is induced by $\text{Dept} d, d.DName pn, V_2 v_2$ and there is a previously unmentioned rewriting found as the subquery induced by $V_1 v_1, \text{Teams} t: R_2: \text{select distinctstruct}(E: t.TMember)$ from $V_1 v_1, \text{Teams} t$ where $v_1.D = \text{"Security"}$ and $v_1.E = t.TMember$ and $v_1.P = t.TProj$.

A subquery SQ of the universal plan is tested for equivalence to $Q$ by chasing both with $\Sigma_C \cup \Sigma_V$ and looking for containment mappings (see section 10.7) between the chase results [PT99].

### 10.4 The C&B Algorithm for Bag Semantics

**Phase 1: Chase.** We ask if there exists a transformation on bag queries that is analogous to the chase of a set query with a constraint $d$ (recall (1) in Section 10.3)? If so, its simplest form would look like this (notice the absence of the distinct keyword):

$$
\text{select } O(m) \\
\frac{\text{select } O(m)}{d} \\
\text{from } \ldots, M m, \ldots \\
\text{where } \ldots \text{ and } B_1 \text{ and } \ldots \\
\text{select } O(m) \\
\text{from } \ldots, M m, N n, \ldots \\
\text{where } \ldots \text{ and } B_1 \text{ and } B_2 \text{ and } \ldots
$$

The analogy stops when we realize that $d$ cannot be just the assertion $\forall(m \in \text{dom } M)[B_1 \rightarrow \exists(n \in \text{dom } N)B_2]$ (recall the notation from section 10.2). For example, if for some $x$ in $\text{dom } M$ for which $B(x)$ holds there are distinct tuple values $y_1,y_2$ in $\text{dom } N$ satisfying $B_2(x,y_1)$ and $B_2(x,y_2)$, then the multiplicity of $O(x)$ in the right hand side of (2) is at least twice its multiplicity in the left hand side. Worse, (2) is not sound even if there is actually a unique value $y$ in $\text{dom } N$ satisfying $B_2(x,y)$, but its associated multiplicity $N[y]$ is greater.

\textsuperscript{7}Implication reduces to membership in the reflexive, symmetric, transitive congruence closure of the equalities in UP, which is computed in PTIME [PT99].
than 1. On the other hand, requiring uniqueness of the value \( y \) in \( \text{dom} \mathcal{N} \) that satisfies \( \mathcal{B}_2(x, y) \), and unit multiplicity for \( y \) will suffice to make (2) sound for all \( O(x) \).

**Chasing with UWDs.** We recognize from the above discussion that the conditions under which transformation (2) is sound are equivalent to the (witness), (functionality) and (multiplicity) constraints (as introduced in definition 10.2.1) for the UWD \( (d) \land (m \in M) \left[ \mathcal{B}_1 \Rightarrow \exists! (n \in \mathcal{N}) \mathcal{B}_2 \right] \). Transformation (2) is hence an example of a new kind of chase, namely that of a bag query with the UWD \((d)\).

The full definition of the chase step with UWDs is more involved than transformation (2) above. It is inspired by the definition of UWDs (10.2.1) and given in section 10.7. We show there that the chase step is a *sound* transformation, i.e. it preserves equivalence under UWDs. Moreover, we show how to use this new chase for *deciding* equivalence of bag queries under UWDs.

The main difference from chasing under set semantics is that, as we chase a bag query \( Q \), we infer that some of its variables are guaranteed to range only over unique tuple values, of multiplicity 1, in any instance satisfying the UWD. We denote these variables with \( \text{UW}(Q) \). Initially, \( \text{UW}(Q) \) is the empty set, but the chase step can update both \( Q \) and \( \text{UW}(Q) \).

**Example.** Recall the example from section 10.1.1. Starting with an empty \( \text{UW}(Q^b) \), \( Q^b \) chases with \( (\text{insider}^b) \) to

\[
Q^b_1 : \quad \text{select} \quad \text{struct}(E : t.\text{TMember})
\]

\[
\text{from} \quad \text{depts} d, d.\text{DProjs} pn, \text{Teams} t, \text{Payroll} p
\]

\[
\text{where} \quad pn = t.\text{TProj} \quad \text{and} \quad d.\text{DName} = \text{“Security”} \quad \text{and} \quad p.\text{PDept} = d.\text{DName} \quad \text{and}
\]

\[
p.\text{Empl} = t.\text{TMember}
\]

and the fact that \( p \) is known to range only over unique witnesses is recorded: \( \text{UW}(Q^b_1) = \{p\} \).

We now focus on the search space for rewritings. Recall from section 10.3 that, for set semantics, the views that are relevant to answering the query are brought into the universal plan as a result of chasing with their chase-in dependencies from \( \Sigma_V \). For bag semantics we only know how to chase with UWDs, which unfortunately do not capture the views in
general. To see why this is the case, consider the view \( V \triangleq \text{select } O \text{ from } M \text{ where } B \) and, analogously to the set semantics case, the chase-in dependency \( c^b_V \triangleq \forall (m \in M) \text{ } B \rightarrow \exists ! (v \in V) \text{ } v = O \). In general, \( c^b_V \) is not satisfied because the unit multiplicity requirement on the tuples bound by \( v \) is not guaranteed to hold, even if \( M \) is a set (just consider that \( O \) projects on some non-key attribute). \( c^b_V \) would hold if \( V \) were defined using \text{select distinct}, as is the case for set semantics. We therefore introduce the concept of chasing a bag query with a bag view:

**Chasing with a view; Universal Plan.** The result of chasing a query \( Q \) with a view \( V \) is the result of chasing \( Q \) with \( (c^b_V) \), without updating \( \text{UW}(Q) \). The formal definition is given in section 10.7, and it contains subtleties on the applicability of this chase step with a view. For the moment, we only note that the effect of chasing with a view is that of bringing it into the universal plan, just like in the set case. We emphasize that since \( (c^b_V) \) is not guaranteed to hold, this transformation is not equivalence-preserving. Still, it enables us to define the search space for scan-minimal rewritings of \( Q \) as being the subqueries of the universal plan obtained by chasing \( Q \) with the semantics constraints expressed as the set of UWDs \( \Sigma_C \), and then with the views. Theorem 10.5.1 shows that for the important class of full UWDs, this universal plan is unique, polynomial in the size of \( Q \), and contains all scan-minimal rewritings.

**Example.** Revisiting the motivating example, we can easily check that the universal plan \( \text{UP}^b \) obtained by first chasing \( Q^b \) with \( (\text{insider}^b) \) to \( Q^b_1 \), and then chasing \( Q^b_1 \) with the views \( V^b_1, V^b_2 \), is the same as \( \text{UP} \) after dropping the \text{distinct} keyword. \( R^b \) and \( R^b_1 \) are subqueries of \( \text{UP}^b \) (there are others, such as \( R^b_2 \), the bag version of \( R_2 \) from the example in section 10.3, which is also a rewriting).

**Phase 2: Backchase.** In this phase, we must find those subqueries of the universal plan which are equivalent to \( Q \). Section 10.3 showed that for set semantics, the equivalence check is reducible to checking equivalence under the constraints capturing the views. While we know how to check equivalence under UWDs (shown below in theorem 10.7.1), remember that bag views are not captured by them.

**Unfolding.** The problem of checking a subquery for equivalence to \( Q \) can nevertheless be reduced to deciding bag equivalence under \( \Sigma_C \), by using the notion of unfolding of \( R \) w.r.t. to the set of views \( V = (V_1, \ldots, V_n) \) defined by the queries \( Q_i \). The result is denoted
unfold_\text{V}(R)$ and obtained as follows: for every variable binding $V_i v_i$ in $R$, (i) rename the variables in $Q_i$ to fresh ones, obtaining $Q'_i$, (ii) construct $R_1$ by substituting $V_i v_i$ in $R$ with $Q_i$’s from clause, (iii) construct $R_2$ by adding $Q_i$’s where clause to $R_1$’s and (iv) for every component $C : expr$ in $Q'_i$’s select clause, substitute $expr$ for $v_i C$ in $R_2$.

**Example.** The unfolding of $R^b$ w.r.t. $V_1, V_2$ is

$$UFR^b : \text{select struct}(E : p_1.\text{Empl})$$

$$\text{from depts } d, d.\text{DProjs } pn, \text{Payroll } p_1, \text{Teams } t, \text{Payroll } p_2$$

$$\text{where } d.\text{DName} = p_1.\text{PDept} \text{ and } t.\text{TMember} = p_2.\text{Empl} \text{ and } d.\text{DName} = \text{"Security"}$$

$$\text{and } pn = t.\text{TProj} \text{ and } p_1.\text{Empl} = t.\text{TMember} \text{ and } d.\text{DName} = p_2.\text{PDept}.$$

The following result is used to check that $R^b$ is indeed a rewriting:

**Proposition 10.4.1** $R$ is a rewriting of $Q$ using the views in $V$ in the presence of the UWDs in $\Sigma_C$ if and only if $\text{unfold}_V(R)$ is equivalent to $Q$ under $\Sigma_C$.

The equivalence of $\text{unfold}_V(R)$ under $\Sigma_C$ is checked by chasing, according to theorem 10.7.1. A detailed example for $R^b$ is deferred until after theorem 10.7.1.

### 10.5 Completeness of the C&B Method

The following justifies why it is enough to restrict our search to the subqueries of the universal plan, for both set and bag semantics. The result shows that the C&B algorithm handles also the general case in which some of the database relations are sets, and some of the views have set semantics (were defined using select distinct). We state the result in its full generality for path-conjunctive (PC) queries and (full) EPCD and UWD dependencies, which are all defined in section 10.7.

We are given a schema with bag and set elements. Let the corresponding semantic constraints be expressed by a set $\Sigma_C$ of full EPCDs, and by a set $\Sigma^b_C$ of full UWDs. Also let $V^b, V^s$ be sets of views with bag, respectively set semantics.

Given a PC query $Q$, we obtain the universal plan $UP$ as follows: if $Q$ has set semantics, interpret all schema elements as sets and the bag views as set views. Capture all views with the EPCDs $\Sigma_V$ and translate all UWDs to their corresponding (witness) and (functionality)
EPCDs $\Sigma_{\text{transl}}$. $UP$ is the result of chasing $Q$ with $\Sigma_C \cup \Sigma_V \cup \Sigma_{\text{transl}}$. If $Q$ has bag semantics, add for every set schema element the “no-duplicates” UWD to $\Sigma^{b}_{\text{nodup}}$. Let $\Sigma^{b}_{\text{implied}}$ be all UWDs implied from $\Sigma_C$ and the “no-duplicate” constraints (as in section 10.2) \(^8\). Capture the set views with the UWDs $\Sigma^{b}_{V_s}$ (recall from section 10.4 that this can be done). $UP$ is the result of chasing $Q$ with $\Sigma^{b}_{C} \cup \Sigma^{b}_{\text{nodup}} \cup \Sigma^{b}_{\text{implied}} \cup \Sigma^{b}_{V_s}$, then with the bag views in $V^{b}$.

**Theorem 10.5.1** All scan-minimal rewritings of $Q$ can be found among the subqueries of the universal plan $UP$, which is unique and polynomial in the size of $Q$.

A particular case for set semantics (no $V^{b}$, or bag schema elements) in the absence of semantic constraints (no $\Sigma^{EPCD}$ or $\Sigma^{UWD}$) was given in [DPT99]. The completeness and decidability results of [CKPS95, LMSS95] follow from our theorem when we restrict it to conjunctive queries in the absence of semantic constraints.

### 10.6 Backchase with Cost-Based Pruning

*Backchase with cost-based pruning* is our algorithm for exploring the subqueries of the universal plan. The algorithm takes as input a query $Q$, a universal plan $UP$ (for the search space), a set of views $V$ and a set of integrity constraints $\Sigma$ and returns the optimal plan for $Q$. It works for all semantics (set, bag, mixed), and has three parameters.

The first parameter is the cost model (procedure $\text{computeCost}$ below), which we don’t further specify, requiring only that it be *monotonic* (recall page 112)\(^8\). The second parameter is a traditional optimizer which, given a rewriting, picks the best execution plan by performing classical index selection and join reordering (procedure $\text{computePlan}$ below). The third parameter is the procedure $\text{areEquivalent}(Q, SQ, V, \Sigma)$, which decides equivalence of query $Q$ and subquery $SQ$ of the universal plan under the views in $V$ and the constraints in $\Sigma$. Recall from section 10.3 that, for set semantics, this equivalence is decided by exhibiting a containment mapping (section 10.7) from $Q$ to the result of chasing $SQ$ with $\Sigma$ and $\Sigma_V$ (the dependencies we use to capture the views). As we shall see shortly in theorem 10.7.1, for bag and bag-set semantics the equivalence is decided by exhibiting

\(^8\)In practice, we do not compute these, but rather achieve their effect by translating UWDs constraints to their witness, functionality and multiplicity EPCDs, so they can interact with $\Sigma_C$. Space does not allow us to elaborate on this subtle point!
special containment mappings between the results of chasing $Q$ and $\text{unfold}_V(SQ)$ with the UWDs in $\Sigma$.

One more bit of notation before showing the algorithm: Each scan over a view $V v$ is uniquely identified by the mapping $h$ used to chase it into the universal plan. We say that the scans in the image under $h$ of $V$’s chase-in dependency are covered by $V v$, and denote with $\text{NUMS}(V v)$ the covered scans whose variables are not it $\text{UW}(UP)$ (we call them non-unit multiplicity scans, which explains the notation). For example, recalling $Q^b_1$ from section 10.4, $\text{NUMS}(V_2 v_2) = \{\text{Teams } t\}$. Recalling our discussion at the end of section 10.4, note that for set and bag-set semantics, all “no-duplicates” UWDs are present, hence there are no non-unit multiplicity scans and $\text{NUMS}(V v)$ is always empty.

**backchase with cost-based pruning**

**input:** query $Q$, universal plan $UP$, views $V$, dependencies $\Sigma$

**param:** areEquivalent, computePlan, computeCost

**output:** cheapest scan-minimal rewriting according to computeCost.

```
for i = 1 to number of scans in $UP$
(1) for each non-pruned subset $S$ of cardinality $i$ of $UP$’s scans
    if $S$ contains a scan over a view $V v$ such that $\text{NUMS}(V v) \cap S \neq \emptyset$
        prune all supersets of $S$ and continue
(2) if $S$ induces no subquery of $UP^b$ continue
    let $SQ$ be the subquery induced by $S$, $plan = \text{computePlan}(SQ)$,
    $cost = \text{computeCost}(plan)$ in
(3) if $plan$ is not cheapest so far, prune all supersets of $S$ and continue
    if areEquivalent($SQ, Q, V, \Sigma$)
(4) record $plan$ and $cost$ as best so far and prune all supersets of $S$

return best plan so far
```

Because the universal plan contains all scan-minimal rewritings among its subqueries and because we explore all non-pruned subqueries (see step (1)), it is enough to show that no pruning step misses the optimal plan. Then completeness of our algorithm follows.

Step (3) performs cost-based pruning, and it is correct only under the monotonic cost assumption. Step (4) prunes only those rewritings that are not scan-minimal, because they already contain the rewriting induced by $S$. The most subtle pruning step is step (2). As discussed above it never applies for set and bag-set semantics. This step rules out
subqueries containing both a scan over a view and some of the NUMS it covers. We omit the proof of why this pruning preserves optimality, but we illustrate it to give some intuition: a subquery of $UP^b$ which is pruned in step (3), is the one induced by $S = \{Teams \ t, \ V_2 \ v_2\}$:

$$SQ: \text{select struct} \ (E : t.TMember) \ \text{from} \ \text{Teams} \ t, \ V_2 \ v_2 \ \text{where} \ v_2.D = \text{"Security"}$$

and $v_2.E = t.TMember$ and $v_2.P = t.TProj$

pruned because $\text{NUMS}(V_2 \ v_2) \cap S = \{\text{Teams}\ \ t\}$. It is easy to see that neither it nor its superqueries can be rewritings, because $SQ$ iterates over $\text{Teams}$ tuples twice (once hidden in the definition of $V_2$) as opposed to only once in $Q^b$. The multiplicities in $SQ$’s result are therefore higher than those in $Q^b$’s and they will be even higher for $SQ$’s superqueries.

It turns out that the subqueries that are not pruned in (2) could alternatively be obtained by “substituting” view scans for the NUMS they cover, in all possible ways. Remarkably, for bag semantics, in the absence of integrity constraints, this operation reduces to that of substituting a view from [CKPS95], and our backchase with cost-based pruning behaves just like their algorithm.

10.7 Formal Details

[PT99] presents the theory of chasing a class of queries with set semantics called pathconjunctive (set PC) with a class of dependencies called embedded path-conjunctive dependencies (EPCDs). In this section, we develop the theory of chasing bag PC queries with UWDs.

Paths. The PC language (formally defined in [PT99]) extends conjunctive queries to complex values and dictionaries, introducing two operations: $\text{dom}\ M$ returns the domain of the dictionary $M$, i.e. the set of its keys, and $M[k]$, is the result of looking up a key $k$ in a dictionary $M$. We define paths as $P ::= x \ | \ c \ | \ SN \ | \ PA \ | \ \text{dom}\ P \ | \ P[x]$ where $x$ stands for variables, $c$ for constants at base types, $SN, A$ for schema, respectively attribute names. Wherever OQL allows a schema name, we allow a set/bag-typed path. The expressions in equalities of the where clause and in the select clause are paths whose type cannot be set/bag or dictionary. [DPT99] shows how dictionaries can model indexes, which we omit here for simplicity sake. For the same reason, all queries in our examples are set/bag PC queries without dictionaries.
**EPCDs and UWDs; Fullness.** EPCDs generalize relational embedded dependencies [AHV95] to our data model, and they have the logical form $\forall (m_1 \in M_1) \ldots (m_k \in M_k) \ B_1 \rightarrow \exists (n_1 \in N_1) \ldots (n_l \in N_l) \ B_2$ where all $M_i, N_j$ are paths, and $B_1, B_2$ are conjunctions of path equalities (same restrictions as in the **from**, respectively **where** clauses of PC queries).

Recalling definition 10.2.1, the (witness), (functionality) and (multiplicity) dependencies are examples for EPCDs 9 UWDs have the same syntax as EPCDs, except for the special quantifiers. We say that an EPCD or a UWD is **full** if $B_1$ implies that all components of the existentially quantified variables (or the variables themselves if not of record type) are equal to paths over the universally quantified variables. Fullness can be checked in PTIME [PT99]. The EPCDs and UWDs in our examples are full, and so are many common integrity constraints.

**Homomorphism.** The definition of the chase step with a UWD relies on the notion of homomorphism: Given UWD $(d)\forall (m_1 \in M_1) \ldots (m_k \in M_k) \ B_1 \rightarrow \exists ! (n_1 \in N_1) \ldots (n_l \in N_l) \ B_2$ and bag PC query $Q = \text{select } O(s_1, \ldots, s_q) \text{from } S_1 s_1, \ldots, S_q s_q \text{where } C(s_1, \ldots, s_q)$, a homomorphism from $d$’s universal part into $Q$ 10 is a mapping $h : \{m_1, \ldots, m_k\} \rightarrow \{s_1, \ldots, s_q\}$ such that (i) $h(M_i) h(m_i)$ is among the bindings of $Q$ for every $i$ and (ii) $h(B_i)$ is implied by $Q$’s conditions $C (C \Rightarrow h(B_i))$. We say that $h$ extends to $d$’s existential part, if it extends to $n_1, \ldots, n_l$ such that (i) and (ii) above hold even when substituting $n_i, N_i, B_2$ for $m_i, M_i, B_1$.

**Chase step.** The chase step’s definition is inspired by definition 10.2.1. The idea is that, as we chase a bag PC query $Q$, we infer that some of its variables are guaranteed to range only over unique tuple values, of multiplicity 1, in any instance satisfying the UWD. We denote these variables with UW($Q$). Initially, UW($Q$) is the empty set, but it can be updated during the chase.

The chase step of $Q$ with $d$ using the homomorphism $h$ from $d$’s universal part is **applicable** if either of $d$’s witness ($w(d)$), functionality ($f(d)$), or multiplicity ($m(d)$) rules below apply. The result of applying the chase step is denoted $\text{chase}_X(Q)$, with $X \in$

---

9This is not accidental: we model bags using dictionaries as well, and exploit this in implementation.

10The notion can be easily adapted to that of homomorphism $h$ from a query $Q_1$ into a query $Q_2$ by considering a dependency whose universal part is constructed from the **from** and **where** clauses of $Q_1$. A homomorphism $h$ from $Q_1$ to $Q_2$ is a containment mapping, if the conditions in $Q_2$ imply the equality of its **select** clause with the image of $Q_1$’s **select** clause under $h$. 120
The functionality and multiplicity rules are necessary: examples can be given in which chasing with the witness rule alone cannot decide equivalence of bag queries. Before showing how the chase with UWDs is used to decide equivalence, we need to generalize the notion of containment mapping from [PT99].

**MP mapping.** Let $Q_1, Q_2$ be bag PC queries. A mapping $h$ from the variables of $Q_1$ to those of $Q_2$ is *multiplicity-preserving* (MP) if and only if (i) $h$ is a homomorphism from $Q_1$ into $Q_2$, (ii) all variables in UW($Q_1$) are mapped into UW($Q_2$), (iii) $h$ is surjective on $Q_2$’s variables that are not in UW($Q_2$), and (iv) for any variables $x, y \in$ UW($Q_1$) with the same image under $h$, the equality $x = y$ is implied by the conditions in $Q_1$’s where clause. A mapping $h$ from $Q_1$ to $Q_2$ is an *MP containment mapping*, if it is an MP mapping and if the conditions in $Q_2$ imply the equality of its select clause with the image of $Q_1$’s select clause.
In particular, if UW(Q₁) is empty and UW(Q₂) contains all of Q₂'s variables, h degenerates to a homomorphism. If both UW(Q₁), UW(Q₂) are empty, h degenerates to a surjective homomorphism and if only UW(Q₂) is empty, there is no MP-mapping from Q₁ to Q₂.

**Theorem 10.7.1 (Bag chase theorem)** Let Q₁, Q₂ be bag PC queries with empty UW(Q₁) and UW(Q₂). Let Σ be a set of UWDs such that the chase of Qᵢ terminates, yielding CQᵢ. Then Q₁ and Q₂ are equivalent under Σ if and only if there exist MP containment mappings h₁₂ : CQ₁ → CQ₂ and h₂₁ : CQ₂ → CQ₁. Moreover, the existence of h₁₂ implies the containment of Q₂ in Q₁ under Σ.

If all UWDs in Σ are full, the chase terminates in polynomial time in the size of Q₁. Still, finding the MP mappings is as hard as graph isomorphism.

Notice that in the absence of constraints we get UW(CQ₁) = UW(CQ₂) = ∅ and we recover the corresponding results of [CV93] extended to PC queries: Q₁, Q₂ are bag equivalent if and only if they are isomorphic, and surjective containment mappings imply containment.

**Example.** We remarked at the end of section 10.2 that in order for Qᵇ to allow the rewriting Rᵇ, we needed the “bag key constraint” (bk) to hold on the Empl attribute of Payroll: ∀(p₁ ∈ Payroll) (p₂ ∈ Payroll) p₁.Empl = p₂.Empl → p₁ = p₂ which is trivially implied by the corresponding key constraint on the underlying set dom Payroll (yet another example of a UWD that is implied from common dependencies on sets). Recall from proposition 10.4.1 that Rᵇ is equivalent to its unfolding

\[ UFRᵇ : \text{select } \text{struct(E : p₁.Empl)} \]
\[ \text{from } \text{depts } d, d.DProjs pn, \text{Payroll } p₁, \text{Teams } t, \text{Payroll } p₂ \]
\[ \text{where } d.DName = p₁.PDept \text{ and } t.TMember = p₂.Empl \text{ and } d.DName = \text{"Security"} \]
\[ \text{and } pn = t.TProj \text{ and } p₁.Empl = t.TMember \]

It therefore suffices to check the equivalence of UFRᵇ and Qᵇ. Qᵇ chases with w(insiderᵇ) and m(insiderᵇ) to Qᵇ₁ with UW(Qᵇ₁) = \{p\}:

\[ Qᵇ₁ : \text{select } \text{struct(E : t.TMember)} \]
\[ \text{from } \text{depts } d, d.DProjs pn, \text{Teams } t, \text{Payroll } p \]
\[ \text{where } pn = t.TProj \text{ and } d.DName = \text{"Security"} \text{ and } p.PDept = d.DName \text{ and } p.Empl = t.TMember \]
$UFR^b$ chases with $w(bk)$, then twice with $m(\text{insider}^b)$ to $Q^b_2$ with $\text{UW}(Q^b_2) = \{p_1, p_2\}$:

$$Q^b_2 : \text{select } \text{struct}(E:p_1.\text{Empl})$$

$$\text{from } \text{depts} \, d, \, d.\text{DProjs} \, pn, \, \text{Payroll} \, p_1, \, \text{Teams} \, t, \, \text{Payroll} \, p_2$$

$$\text{where } d.\text{DName} = p_1.\text{PDept} \, \text{and } \, t.\text{TMember} = p_2.\text{Empl} \, \text{and } \, d.\text{DName} = \text{"Security" }$$

$$\text{and } pn = t.\text{TProj} \, \text{and } \, p_1.\text{Empl} = t.\text{TMember} \, \text{and } \, p_1.\text{PDept} = p_2.\text{PDept}$$

We exhibit MP containment mappings $h_{2,1} : Q^b_2 \rightarrow Q^b_1$ and $h_{1,2} : Q^b_1 \rightarrow Q^b_2$ as the identity mapping on $d, pn, t$ extended with $\{p_1 \mapsto p, p_2 \mapsto p\}$, respectively $\{p \mapsto p_1\}$. Notice that, in the absence of $(bk)$, $\text{UW}(Q^b_2)$ would contain only $p_1$, and $h_{1,2}$ wouldn’t be an MP mapping because of violating condition (ii) in the definition: $p_2$ is not in $h_{1,2}$’s image. $R^b$ would therefore not be a rewriting of $Q^b$. 

By chasing with the UWDs in $\Sigma$ and the “no-duplicates” UWDs (full as well!) for every relation in the schema, we decide bag-set equivalence under $\Sigma$, thus extending the corresponding result of [CV93].

**Chasing with a bag view.** We define here the concept of chasing with a view, introduced in section 10.4. A **view mapping** from a bag PC view $V$, into a bag PC query $Q$, is a homomorphism $h$ from the universal part of $V$’s chase-in dependency $(c^b_V)$, which is injective on the variables not mapped into $\text{UW}(Q)$. The **result** of the **view chase step** of query $Q$ with $V$ using $h$ is $\text{chase}_{w(c^b_V)}(Q)$, the result of chasing $Q$ using $h$ with the witness rule for $(c^b_V)$ (not surprisingly, the functionality and multiplicity rules are disabled since the corresponding statements are not true). However, as opposed to the regular witness rule, the chase step with $V$ is **applicable** as long as it wasn’t previously performed using the same view mapping $h$. 

123
10.8 Extension to Grouping Views

Assume that instead of $V_1$ from our motivating example, a more storage-efficient nested view $N_1$ is materialized. Then instead of $R$, we obtain the equivalent rewriting $NR$.

$$N_1: \text{select} \ \text{distinct} \ \text{struct}(D: d.DName, G: \text{select} \ \text{distinctstruct}(P: pn, E: p.Empl) \ \text{from} \ d.DProjs, Payroll p \ \text{where} \ d.DName = p.PDept) \ \text{from} \ \text{depts} \ d$$

$$NR: \text{select} \ \text{distinctstruct}(E: m.E) \ \text{from} \ N_1 n, n.G m, V_2 v_2 \ \text{where} \ n.D = \text{"Security" and } m.P = v_2.P \ \text{and} \ m.E = v_2.E \ \text{and} \ n.D = v_2.D$$

When trying to capture $N_1$ by constraints, here is the one corresponding to the chase-in dependency:

$$(c_{N_1}) \ \forall (d \in \text{depts}) \ \exists (n \in N_1) \ d.DName = n.D \land \forall (pn \in d.DProjs) (p \in \text{Payroll}) d.DName = p.PDept \rightarrow \exists (m \in n.G) m.P = pn \land m.E = p.Empl$$

This is not an EPCD (it has more than one alternation of quantifiers) and we don’t know how to chase with it. In general, nested views cannot be captured by EPCDs. We can nevertheless prove that this capture is possible whenever the flat typed attributes of the output tuples ($d.DName$ for $N_1$) functionally determine all of the inner query’s path expressions ($d.DProjs$ and $d.DName$) involving its free variables ($d$). We omit a formal definition, but we exemplify on $N_1$, for which we ask that the functional dependency below be implied by the integrity constraints (the key on $DName$ for class $Dept$ in this case):
\[(FD_{N1}) \forall(d_1 \in \text{depts})(d_2 \in \text{depts})d_1.DName = d_2.DName \rightarrow d_1.DProjs = d_2.DProjs \land d_1.DName = d_2.DName\]

We call queries whose associated functional dependency \(FD\) is satisfied grouping queries, because they contain as particular case all SQL-like queries with a group-by clause (but which, against SQL syntax, do not aggregate the groups), for whom \(FD\) is trivially satisfied. In general, we decide whether a query is grouping by using the chase to decide \(FD\)’s implication [PT99].

The reader can check that \(c_{N1}\) is equivalent to \(\{c_{N1}^d, c_{N1}^i\}\) below, but only in the presence of \(FD_{N1}:\)

\[
\begin{align*}
& (c_{N1}^d) \forall(d \in \text{depts}) \exists(n \in N_1) d.DName = n.D \\
& (c_{N1}^i) \forall(d \in \text{depts})(n \in N_1)(pn \in d.DProjs)(p \in \text{Payroll}) \\
& \quad d.DName = n.D \land d.DName = p.PDept \rightarrow \exists(m \in n.G)m.P = pn \land m.E = p.Empl
\end{align*}
\]

Now \(\{c_{N1}^d, c_{N1}^i\}\) chases \((n \in N_1)\) and \((m \in n.G)\) into the universal plan, which contains \(NR\) as a subquery.

**Theorem 10.8.1** The C&B algorithm remains complete even when we add grouping views with bag and set semantics, generalized to arbitrarily many nesting levels. Moreover, we can decide equivalence of set (bag) grouping queries in the presence of full EPCDs (UWDs).

### 10.9 Conclusions and Future Work

This work reports on several extensions of the chase & backchase (C&B) method for optimizing queries with materialized views and integrity constraints. The theoretical contributions include the extension of the method to bag semantics, as well as to a class of grouping views. We also give completeness theorems for both set and bag case. On the practical side, we have integrated the C&B method with a standard cost-based optimization method. The experimental results show a great improvement in performance over the case when the two methods are implemented as separate phases. The resulting optimization algorithm is the only systematic cost-based optimizer, that works for both sets and bags, takes advantage of both materialized views and integrity constraints, and has precise
optimality guarantees, that we are aware of.

**Future Work.** We would like to extend our comprehensive method to include, in a systematic way, queries and views with common aggregates and union, as well as grouping queries. On the practical side, we plan to investigate to what extent the cost-based chase & backchase algorithm can be implemented on top of an already existing commercial optimizer.
Chapter 11

Related Work and Conclusions

11.1 Related Work

XML storage Early research explored the obvious avenue of eliminating heterogeneity by storing all XML data relationally, with the added benefit of exploiting the mature relational technology. The emerging consensus is that, while relational storage can (and should) be used in many important cases, it is not always sufficient. This is because highly unstructured XML is harder to store "meaningfully" in relations, therefore sometimes we may have to store only part of the XML, keeping the rest as is ([DFS99] only used relations for the frequently encountered XML subtree types, and stored the outliers in overflow graphs.) [STZ+99] used the DTD information to generate the relational storage schema, and therefore faces the problem that DTDs are too permissive to fully capture relationally in a natural way. Moreover, as the study in [Sah00] reveals, DTDs are often misused, under-specifying the document structure for the sake of simplicity. Of course, there is always the generic mapping storing the edge relations of the XML tree in RDB tables ([FK99]). In this scheme however, a path of $n$ navigation steps translates into just as many joins of the edge relations, thus simulating the graph navigation that is typical of DOM-based systems and therefore taking only limited advantage of the RDBs set-oriented processing capabilities (by allowing join reordering, which is impossible in DOM-based navigation). Conclusion: mixed storage is needed.
**Existing reformulation algorithms.** In general, in the public-as-view approach, given the view \( V \), reformulating a client query \( Q \) posed against the published schema to an executable query \( X \) posed against the proprietary storage schema involves composing \( Q \) with \( V \), \( X = Q \circ V \) and the class of algorithms performing this task is called *composition-with-views*. This task is straightforward if everything is relational and no integrity constraints are allowed, but quite complex in XML publishing of relational data [FTS00, SKS+01], and in a semistructured data context [PGMW95]. In the complementary storage-as-view approach, the problem is challenging even in a purely relational setting. It amounts to solving the equation \( Q = X \circ W \) for \( X \), where \( W \) is the view defining the storage in terms of the published schema. The class of algorithms performing this task is called *rewriting-with-views* and is used in [DFS99, LRO96, MFK01].

**Existing systems** Some existing systems follow the public-as-view-only approach. For XML publishing, and when the storage schema is purely relational, our system subsumes the expressive power of XPeranto [CKS+00, SKS+01] and SilkRoute [FTS00, FMS01]. For the storage-as-view-only approach, we handle as particular instances XML publishing as in Agora [MFK01] and STORED [DFS99] and purely relational integration as in the Information Manifold [LRO96].

**XPeranto** is most suited when there is no redundancy, no mixed storage, and the relational source is unique. The public-as-view mapping is specified by using a default encoding of the relations as XML, and writing an XQuery against this. As a consequence of the public-as-view-only mapping, this system cannot integrate \( RView \), \( XView \), Vienna.xml from the motivating example. MARS borrows one of the tagging approaches proposed in XPeranto.

**SilkRoute** This system is applicable whenever XPeranto is, and shares with it the benefits and drawbacks of the public-as-view-only approach to schema mapping. A unique feature of its mapping language, RXL, is that allows Skolem functions explicitly as language primitives. They can express the outer joins and nested correlated queries of XPeranto, and even mappings that go beyond the latter. We capture them with constraints just like we did for XQueries in section 5.2. We favor an RXL-like syntax for expressing schema mappings.
Agora allows one to query mixed relational and XML sources, by specifying mappings in storage-as-view-only style, and thus it inherits the limitations of this approach, not being applicable when the public schema hides proprietary data fields. The system uses a relational encoding of XML (based on a child and descendant relation, similar to MARS), and sources are specified as SQL views against this encoding. Agora’s reformulation algorithm is based on a relational algorithm for rewriting-with-views, in the absence of integrity constraints, but extended to capture some of the XML-specific behavior (such as that a child has precisely one parent, etc.). There is no theoretical guarantee however that these extensions result in a complete algorithm. Such a result means proving that the extended algorithm finds all reformulations which hold in all XML documents, but not in all possible interpretations of the child and descendant relations. There are plenty such examples: desc-child=child-desc is a very simple one, which happens to be handled by Agora, but what about other, less obvious ones?

Stored is a system whose goal is to store XML losslessly, using relational storage. It defines relational views of the highly structured XML fragments, and it uses an overflow graph for the unstructured one. The queries are reformulated to mixed queries over the relations and the XML overflow graph, making the problem of query reformulation an instance of a MARS application. Remarkably, the query reformulation algorithm in Stored performs rewriting-with-views by reduction to composition-with-views. The reduction uses the idea of view inversion introduced in [DG97]. This technique only works because the views are restricted to use only simple navigation (most salient restriction: no descendant navigation), and it does not take into account any integrity constraints. Nevertheless, this approach provided us with a first indication that the problems of rewriting-with-views and composition-with-views are unifyable.

Information Manifold was the first system proposing storage-as-view-style mappings, in data integration. Notably, the drawback of this mapping style of not being able to hide proprietary data was not an issue in the Information Manifold, which was modeling web information sources that were typically queried by filling out forms. In this scenario, each source is one form, all of whose fields are visible, so it can always be described as lossless view over the public schema. In this last case the C&B algorithm has the same complexity as the Minicon [PH01] complete algorithm, NP in the size of the query, which
was shown to be optimal in [LMSS95].

The problem of rewriting regular path queries with inverse (RPQIs) with RPQI views in a storage-as-view semistructured data context was addressed in [CGLV99, CGLV00b]. However, the main technical difficulties we have solved for the translation (see above) are in XQuery but not in RPQs.

Reformulation under constraints None of the algorithms we just surveyed allow for additional integrity constraints on the schemas, as our approach does. [GL01] and [TSI94] propose algorithms which do take into account some constraints (e.g., referential integrity constraints) and they run in PTIME. The disadvantage here is missing rewritings (unless P=NP, since the problem is NP-hard).

Incomplete information. A related line of work in relational data integration, has a different goal: finding a maximally contained reformulation when the views contain incomplete information (the so-called sound view assumption). In the XML publishing scenarios we consider, we need exact reformulation under the exact view assumption. [FLM99] uses an idea from [DG97] to reduce the schema correspondence given in the combined LAV and GAV approaches (GLAV) to a pure GAV correspondence. Each client query has a single reformulation, which is a highly redundant logic program. The technique does not apply to our publishing scenario because the logic program accesses all sources containing relevant information and thus defeats the purpose of redundant storage. Moreover, no constraints are taken into account. [CGL01] extends these ideas to allow for a restricted class of constraints on the global/published schema, which can be translated to further logic program rules and added to the reformulation program. In a semistructured context, [CGLV00b] solves the problem completely for RPQI queries and RPQI views. [CGLV00b, CGLV01] also study the related problem of answering conjunctive RPQIs using views.

Minimization under constraints. According to our new completeness result, the C&B algorithm is a complete algorithm for minimization of (unions of) conjunctive queries under disjunctive embedded dependencies. The original paper on minimization of conjunctive queries in the absence of dependencies is [CM77]. [ASU79] lists as an open problem even the special case of the minimization of an SPJ query under functional dependencies. [Gry98] minimizes conjunctive queries under inclusion dependencies. All of these (and more general cases) are solved by the C&B algorithm.
11.2 Conclusions

The conceptual contribution of this work to the XML publishing research topic is a uniform
solution to the problem of finding minimal reformulations of XQueries, under schema
correspondences that combine the public-as-view and storage-as-view approaches, where
the views are expressed in XQuery, under mixed storage, RDB and XML, and under
additional integrity constraints on both the public and storage schemas.

The nature of the challenges in doing so is both descriptive (providing a language
rich enough to model the schema correspondence) and computational (finding the optimal
reformulation, given that redundancy enables several of them).

Expressing the schema correspondence. Constraints turn out to be a very expressive
formalism, allowing us to specify the mappings supported by existing systems, and beyond.
The direction-neutrality of constraints allows us to simultaneously specify public-as-view
and storage-as-view mappings in the schema correspondence.

A remarkable side benefit of this approach is that it can handle in the same unified way
redundant storage (typical in XML applications), constraints in XML data (as specified
by XML Schema) and constraints in the relational storage.

Taking constraints into account is not just a choice in our approach, but rather and
approach-independent necessity; constraints arise naturally in the relational storage of
XML, as a consequence of the fact that the XML data model is a tree.

The reformulation algorithm. We have shown an algorithm for the reformulation of
client XQueries in XML publishing scenarios, when the correspondence between public and
storage schema is given by XQuery views in both directions and integrity constraints are
present on both schemas. The algorithm is complete and asymptotically optimal for an
expressive class of client query and views (behaved XQueries) and XML integrity constraints
(bounded XICs). We have shown that even modest extensions of the class of views or XICs
result in incompleteness of reformulation, giving a boundary of how far we can push the
technique.

A matter of independent interest is the fact that our chase-based reformulation algo-

rithm achieves the combined effect of existing algorithms for composition-with-views and
Practicality of the approach. We have built a system called MARS implementing the C&B algorithm and the optimizations discussed in this thesis. This entailed a good deal of engineering. Our experimental evaluation shows that the algorithm is practical, i.e. feasible and worthwhile, which is not a priori self-understood, given that many of the problems we tackle here are NP-hard or worse.
Chapter 12

Directions for Future Work

We mention here two interesting directions which we consider promising for successful application of the C&B algorithm. Section 12.1 describes some incipient work on reformulation of XML constraints, and section 12.2 gives an idea on using the C&B algorithm to guarantee data security.

12.1 XML Constraint Reformulation

In the query reformulation problem we allowed for both the public and the storage schema to be subject to unspecified integrity constraints. Recall however that the published data is virtual. Before guaranteeing that the published data satisfies any integrity constraints, the DB administrator must be sure that these follow from the schema correspondence and the storage constraints. When applicable, the translations that we develop for query reformulation can help with this.

Reformulation versus testing implication. Indeed, the schema correspondence and the storage constraints can be compiled into a set $\Delta$ of relational constraints while the desired XML constraints on the published schema can also be compiled into a set $D$ of relational constraints. We then ask if $\Delta \models D$ and we use the chase to test it. If it checks out, we are happy. But what if it does not? To ensure some badly desired constraint $d$ on the published data we may be in a position to enforce additional constraints on the

\textsuperscript{1}There are many common applications where the chase with such dependencies is guaranteed to terminate. We can check this using the condition in section C.4.
storage data. But which ones? We could guess, and test again as above, but there may be a better way. Namely, ignoring for the moment the storage constraints, to reformulate \(d\) into a storage constraint \(\delta\) that is equivalent to \(d\) modulo the schema correspondence. It may then be easier to redesign the storage constraints in order to enforce \(\delta\) (hence \(d\)).

We present a way to reuse any query reformulation algorithm for constraint reformulation, exploiting the following fundamental reduction between query containment and constraint satisfaction.

**Proposition 12.1.1**  
(a) For every XIC \(d\) there are XBind queries \(Q_1^d, Q_2^d\) such that for any instance \(I, I \models d \iff Q_1^d(I) \subseteq Q_2^d(I)\). (b) For every XBind queries \(Q_1, Q_2\), there is an XIC \(\text{cont}(Q_1, Q_2)\) such that for every instance \(I, Q_1(I) \subseteq Q_2(I) \iff I \models \text{cont}(Q_1, Q_2)\).

**Proof:** (a) For \(d\) of form \(\forall x [B(x) \rightarrow \exists y C(x, y)]\), construct \(Q_1^d(x) \leftarrow B(x)\) and \(Q_2^d(x) \leftarrow B(x) \land C(x, y)\).

(b) For \(Q_1(x) \leftarrow B_1(x, y)\) and \(Q_2(x) \leftarrow B_2(x, z)\),
\[
\text{cont}(Q_1, Q_2) = \forall x \forall y [B_1(x, y) \rightarrow \exists z B_2(x, z)].
\]

This result suggests the following algorithm:

**XIC reformulation algorithm.**

1. construct \(Q_1^d, Q_2^d\),
2. reformulate each against \(S \cup \text{Aux}\), to \(R_1\), resp. \(R_2\)
3. construct \(\text{cont}(R_1, R_2)\)
4. return the restriction of \(\text{cont}(R_1, R_2)\) to \(S\)

Since in general \(d\) quantifies over XML nodes (see the XIC (1.1) in example 1.1.2), \(Q_1^d, Q_2^d\) cannot be reformulated against the storage schema \(S\) only, as it does not contain these nodes. On the other hand, \(Q_1^d, Q_2^d\) are XBind queries, which, if they have a reformulation plan, can be reformulated against \(S \cup \text{Aux}\) (recall Chapter 6). By the following result, we can always turn \(\text{cont}(R_1, R_2)\) (against \(S \cup \text{Aux}\)) into a co-satisfiable dependency formulated solely against \(S\):

**Proposition 12.1.2** Let \(d_R\) be obtained from \(\text{cont}(R_1, R_2)\) by simply dropping all atoms involving any variable \(x\) appearing as the result of a function from \(\text{Aux}\). Then on all
instances satisfying the schema correspondence, \( \text{cont}(R_1, R_2) \) is satisfied if and only if \( d_R \) is.

**Example**  Let \( d \) be the XIC (1.1) from example 1.1.2. Then the XBind query \( Q^d_1 \) (as in proposition 12.1.1) is

\[
Q^d_1(e_1,e_2,n) \leftarrow //\text{empl}(e_1),[./\text{name}/\text{text}](e_1,n)
//\text{empl}(e_2),[./\text{name}/\text{text}](e_2,n)
\]

and \( Q^d_2 \) has an additional equality atom \( e_1 = e_2 \).

Observe that \( Q^d_1, Q^d_2 \) cannot be reformulated against the relational schema \( \{ P, E \} \), because this does not contain the XML nodes \( e_1, e_2 \), which are invented by the publishing query. However, we can reformulate the queries over schema \( S \cup \text{Aux} \). Let \( F_e \) be the function inventing the new empl nodes, and denote with \( G_e \) its graph. The reformulation yields

\[
R_1(e_1,e_2,n) \leftarrow P(n,sn_1),E(sn_1,sl_1),G_e(e_1,n,sn_1,sl_1),
P(n,sn_2),E(sn_2,sl_2),G_e(e_2,n,sn_2,sl_2)
\]

and \( R_2 \) has the additional equality \( e_1 = e_2 \), as well as \( sn_1 = sn_2, sl_1 = sl_2 \) (implied by the injectivity of \( F_e \)). Then \( \text{cont}(R_1, R_2) \) is (after eliminating from the conclusion of the implication all atoms appearing in its premise):

\[
\forall n \forall sn_1 \forall sn_2 \forall sl_1 \forall sl_2 \forall e_1 \forall e_2 \quad [ P(n,sn_1) \land E(sn_1,sl_1) \land G_e(e_1,n,sn_1,sl_1) \land
P(n,sn_2) \land E(sn_2,sl_2) \land G_e(e_2,n,sn_2,sl_2) \rightarrow e_1 = e_2 \land sn_1 = sn_2 \land sl_1 = sl_2]
\]

By proposition 12.1.2, \( \text{cont}(R_1, R_2) \) is satisfied iff \( d_R \) below is:

\[
\forall n \forall sn_1 \forall sn_2 \forall sl_1 \forall sl_2 \quad [ P(n,sn_1) \land E(sn_1,sl_1) \land P(n,sn_2) \land E(sn_2,sl_2) \rightarrow sn_1 = sn_2 \land sl_1 = sl_2]
\]

135
which is expressed solely against \( \{P, E\} \). Observe that we have translated a key constraint to a key constraint on the join of \( P \) and \( E \). this is a non-standard constraint, which won’t be automatically enforced by the RDBMS. However in the presence of the following foreign key constraint ("all persons are employed"):

\[
\forall n \forall ssn \; [P(n, ssn) \rightarrow \exists sal \; E(ssn, sal)]
\]

\( d_R \) could be further simplified to obtain a real key constraint on the name attribute of \( P \):

\[
\forall n \forall s n_1 \forall s n_2 \; [P(n, s n_1) \land P(n, s n_2) \rightarrow s n_1 = s n_2]
\]

This is an example where we need to take into account relational foreign key constraints in order to reformulate XML key constraints into key constraints. The benefit is that the enforcement of the latter is easily delegated to the underlying RDBMS.

**Future Work**  Notice that by “plugging in” any sound query reformulation algorithm into this approach, we obtain a sound algorithm for constraint reformulation. However, there is no guarantee that this results in a complete constraint reformulation algorithm, even when the chase is known to terminate. This is because either or both of \( Q_1^d, Q_2^d \) may have no reformulation, while nevertheless there exists a constraint on \( S \) that is co-satisfiable with \( d \). We are interested in searching for a complete reformulation algorithm.

### 12.2 Data Security in XML Publishing

One of the goals in data publishing is to publish various virtual views of the same proprietary data as needed by the interaction with different business partners. Another goal, which is a cross-purposes with the above, is to hide certain proprietary data, or correlations between proprietary data items. Once the set of published views grows sufficiently, it becomes non-trivial to guarantee that there is no query over the set of public views which retrieves a certain proprietary data correlation that was meant to be hidden.
Therefore we are interested in tools which, given a secret data correlation and a set of public views over the proprietary data, can check whether the correlation is not exposed.

**Reducing the problem to deciding the existence of a reformulation.** Here is how we cast this decision problem as the problem of checking the existence of a reformulation: Express the secret data correlation as the result of an *exposing* query $E(S)$ over the proprietary storage schema $S$. Clients are not allowed to see this result. Note that no client can formulate such a query. Instead, he could try to formulate an *intrusive* query $I(P)$ against the published schema $P$ such that $I$ returns the same answer as $E$. We want our tool to guarantee that there exists no such $I$, or to show a counterexample otherwise.

**Example** We illustrate a simple scenario based on example 1.1.1 from Chapter 1. Assume that clients are allowed to see the names of all patients, as well as the names of all ailments, but not the correlation telling us what ailments each patient potentially suffers from. Suppose however that we have published the correlation between patients and physicians they see, as well as that between physicians and the diseases they are specialized in treating (see Figure 12.1). A client could now join the two correlations to obtain the secret information.

\[\text{Figure 12.1: One Aspect of Data Security}\]

**Checking data security** Here is how we detect such a case: we apply the C&B reformulation algorithm in the *opposite* direction, to reformulate $E(S)$ from the storage schema...
to some $I(P)$ against the public schema.\textsuperscript{2} If such a reformulation is found, we have a proof that the client can retrieve the secret answer. But if no reformulation is found, then we only have a guarantee that the data is secure if the reformulation algorithm is \textit{complete}, i.e. if a minimal reformulation exists, it will be found. Since any reformulation has at least one corresponding minimal reformulation, the overall algorithm gives us no false negatives when it is complete.

\textbf{Future Work} As we have shown in this thesis, the completeness of the reformulation algorithm depends crucially on the expressivity of the language used by the client to query and by the database administrator to specify the schema correspondence. While we were able to establish completeness for the behaved fragment of XQuery, we know that our algorithm turns incomplete when extending XQueries with certain primitives (such as wildcard child navigation). We would like to investigate extensions of the C&B algorithm that guarantee completeness for larger XQuery fragments.

\textsuperscript{2}The fact that our algorithm handles both public-as-view and storage-as-view simulatneously is crucial for the feasibility of this step.
Bibliography


147

Appendix A

An XQuery Primer

The formal semantics of XQuery is standardized by the W3C recommendation [W3Ce]. Sections A.2 and 5.2 provide alternative formal semantics based on relational conjunctive queries, respectively First Order Logic. In this section, we give an informal overview of XQuery, leaving out most bells and whistles.

The following grammar defines the fragment of XQuery which our techniques can handle (here $n$ is any tag or attribute name, $v$ any variable name, and $s$ any string constant):

\[
\begin{align*}
\text{query} & ::= \text{`for'} \text{bindings where conditions return clause} \\
\text{bindings} & ::= \text{binding} | \text{binding',' bindings} \\
\text{binding} & ::= \text{var 'in' path} \\
\text{path} & ::= \text{var sep p | 'document('}s')\text{'}sep p \\
\text{(variable) var} & ::= \text{'}$v' \\
\text{(separator) sep} & ::= \text{'/'} | \text{'//'} \\
\text{(path) p} & ::= p_1|p_2 | p_1 \text{ sep } p_2 | p[q] | . . | * | n | @n | @* | \text{text()} | \text{id(p)} | \text{id(s)} \\
\text{(qualifier) q} & ::= q_1 \text{ and } q_2 | q_1 \text{ or } q_2 | p \text{ | } p = s | p_1 = p_2 | p_1 == p_2 | p \neq s | p_1 \neq p_2 | p_1 \neq p_2 \\
\text{conditions} & ::= \text{condition} | \text{condition'and' conditions} | \text{condition'or' conditions} \\
\text{condition} & ::= \text{exists'} \text{var 'in' path 'satisfies' condition} \text{ | path = s} \text{ | path}_1 = \text{path}_2 \\
& \quad | \text{path}_1 == \text{path}_2 \text{ | path \neq s} \text{ | path}_1 \neq \text{path}_2 \text{ | path}_1 \neq \text{path}_2
\end{align*}
\]

To describe the meaning of XQueries, recall that an XML document represents a rooted tree whose nodes include elements and attributes [W3Ca]. Attributes are attached to
elements, and so are text strings.

An XQuery is evaluated in two stages: in the first stage, which we shall call the binding stage, all variables appearing in the for clause are bound to values in the XML tree if they satisfy the conditions in the where clause. The result of the binding stage is the list of tuples of values the variables were bound to. In the second stage, which we call the tagging stage, new XML trees are output, one for each tuple of variable bindings, using the XML template provided by the return clause and instantiating it with the tuple of variable bindings. We do not show a grammar for the return clause, but rather refer the reader to the examples in section 9.1.

A variable $v$ is introduced by a binding expression $v \text{ in } p$ saying that $v$ is bound to all values in the XML tree reachable by navigation according to the XPath $p$. Therefore, the meaning of XQueries is based crucially on that of XPath expressions.

**Meaning of XPath expressions.** The meaning of XPath expression is standardized in [W3Cb] and formalized in [Wad99]. We give a brief description below: given a context node $x$ in the document tree, the meaning $[p]x$ of an XPath expression $p$ is the set of nodes that can be reached from $x$ when “navigating” the tree according to $p$. This set is ordered according to the order in which the nodes appear in the document. One needs to explain how navigation composes for the various operators that appear in the XPath definition. For example, $[p_1/p_2]x$ is the union of all $[p_2]y$ for all $y$ that are element children of the nodes in $[p_1]x$. Similarly for $[/p_2]x$ except $y$ ranges over the element children of the root node determined by the context ($x$ is not used!). For $p_1//p_2$ and $//p_2$ replace “children” with “descendants”. Moreover, we have $[.]x = \{x\}$ while $[n]x$ and $[@n]x$ consist of the element children, respectively attributes, of node $x$ that have tag, respectively name, $n$. $[*]x$ and $[@*]x$ are the children, respectively attributes of node $x$, regardless of their name. $[..]x$ is the parent of node $x$. $[\text{text()}]x$ denotes all strings attached to the element node $x$. Finally, $[p[q]]x$ selects those nodes $y$ in $[p]x$ such that $q y$ holds true. The meaning of the qualifier $q$ is a boolean value, obtained as follows: $[p]q x$ is true if $[p]x$ is not empty. $[p = s]q x$ is true if $s \in [p]x$.

**Equality and inequality.** These predicates are defined in a rather non-traditional way: $[p_1 = p_2]q x$ is true if some node returned by path $p_1$ is equal to some node returned
by path $p_2$. $[p_1 \neq p_2]_q$ is true if some node returned by path $p_1$ is not equal to some node returned by path $p_2$ [W3Cd]. Note that inequalities are not equivalent to the negation of their corresponding equality. In particular, if $p_1$ evaluates to the empty set of elements and $p_2$ is non-empty, both $[p_1 \neq p_2]_q$ and $[p_1 = p_2]_q$ are false! Equality tests must of course typecheck, and they are satisfied for text and attribute nodes if and only if the string values are equal. There has been a long standing debate over the meaning of equality of elements in XQuery/XPath. The consensus that seems to emerge is that there are two kinds of element equality: by node identity and by value. In the first kind, denoted ’==$’, an element node is only equal to itself. This definition of element equality by identity follows XML-QL [DFF+99]. The second kind, denoted ’=’, holds of two elements if and only if their corresponding trees are isomorphic (as opposed to the original, ad-hoc treatment in [W3Cb]).

There is much more about XPath and XQuery semantics that can be found in [W3Cb, Wad99, W3Ce]. Additional operators are described below. As we can see, inspired loosely by path specification in UNIX-like file directory systems, XPath was designed to be a (1) compact and (2) expressive notation. Its full-fledged definition has many features inspired by practical considerations. The techniques that we bring to bear in this chapter can tackle many of the features of full-fledged XPath, but not all (see 9.4). Similarly, XQuery allows for many other query features, such as aggregates, universal quantification in the where clause, user-defined functions, which our techniques do not help with. As we shall see shortly, this does not mean that we cannot reformulate the richer queries, it only means that we have no theoretical guarantee that our reformulation algorithm is complete in that case, i.e. that it can find a reformulation whenever it exists.

### A.1 The Behaved and the NP Fragments of XQuery

#### Well-behaved XQueries.
For our completeness result, we are be interested in the fragment of well-behaved XQueries defined by substituting the following productions for the corresponding ones in the grammar from section A.

\[
(path) \ p \ ::= \ p_1 | p_2 | p_1 \ sep \ p_2 | p[q] | . | n | @n | @* | \text{text()} | \text{id}(p) | \text{id}(s) \quad (A.1)
\]
Note that we have removed the parent navigation step .. and the wildcard child navigation step * from path expressions.

NP XQueries. NP XQueries are a restriction of the Well-behaved fragment, from which the following primitives are eliminated:

1. disjunction (| in paths, or in qualifiers and conditions
2. equality by node identity (both in qualifiers and conditions)
3. all inequalities

As the name suggests, containment of NP XQueries lies in NP. It turns out that we cannot extend this fragment without raising the complexity of containment to at least $\Pi^p_2$ (see the results in chapter 9).

Given the fact that adding the union operator or inequalities to conjunctive queries results in a jump in the complexity of containment from $NP$ to $\Pi^p_2$, we are not surprised that the same effect is observed when adding disjunction and inequalities to the NP fragment of XQuery. This is why we call the fragment of XQuery that parallels the behavior of conjunctive queries behaved.

It is surprising however that adding the wildcard child navigation to the $NP$ fragment also raises the complexity of containment to $\Pi^p_2$, and similarly when adding equality by node identity.

A.2 Compiling XBind Queries to Unions of Conjunctive Queries

XBind queries are compiled to (unions of) conjunctive queries, against the relational signature of GReX. This translation is performed by first translating away the disjunction (| in paths, or in qualifiers), thus obtaining a union of simple, disjunction-free expressions:

$/(son|daughter)$ translates to $/son \cup /daughter$. Next, we translate each of these according to the operator $T()$ operator (refer to the grammar given in appendix A). $z, u$ below denote fresh variables for every application of a rule in which they appear.

$$P(x,p_1/p_2,y) = P(x,p_1,z) \cup P(z,p_2,y)$$
\[ \mathcal{P}(x, p_1 / p_2, y) = \mathcal{P}(x, p_1, z) \cup \{ \text{desc}(z, u) \} \cup \mathcal{P}(u, p_2, y) \]
\[ \mathcal{P}(x, p[q], y) = \mathcal{P}(x, p, y) \cup \mathcal{Q}(y, q) \]
\[ \mathcal{P}(x, .., y) = \{ x = y \} \]
\[ \mathcal{P}(x, n, y) = \{ \text{child}(x, y), \text{tag}(y, "n") \} \]
\[ \mathcal{P}(x, *, y) = \{ \text{child}(x, y) \} \]
\[ \mathcal{P}(x, .., y) = \{ \text{child}(y, x) \} \]
\[ \mathcal{P}(x, @n, y) = \{ \text{attr}(x, "n", y) \} \]
\[ \mathcal{P}(x, @*, y) = \{ \text{attr}(x, z, y) \} \]
\[ \mathcal{P}(x, \text{text}(), y) = \{ \text{text}(x, y) \} \]
\[ \mathcal{P}(x, \text{id}(p), y) = \mathcal{P}(x, p, z) \cup \{ \text{id}(z, y) \} \]
\[ \mathcal{P}(x, \text{id}(s), y) = \{ \text{id}(s, y) \} \]
\[ \mathcal{Q}(x, q_1 \text{ and } q_2) = \mathcal{Q}(x, q_1) \cup \mathcal{Q}(x, q_2) \]
\[ \mathcal{Q}(x, p) = \mathcal{P}(x, p, z) \]
\[ \mathcal{Q}(x, p = s) = \mathcal{P}(x, p, s) \]
\[ \mathcal{Q}(x, p_1 = p_2) = \mathcal{P}(x, p_1, z) \cup \mathcal{P}(x, p_2, z) \text{ if } p_1, p_2 \text{ not of element type} \]
\[ = \mathcal{P}(x, p_1, z) \cup \mathcal{P}(x, p_2, u) \cup \{ \text{copy}(z, u) \} \text{ otherwise} \]

For brevity, we write \( p \) has element/string type but we mean the type of \( p \) is set of element node identities/strings. It is not hard to see that this translation captures exactly the formal semantics of XPath [Wad99] over models in which desc has the intended interpretation.
Appendix B

Formal Details and Proofs for Chapter 4

B.1 Proof of Theorem 4.2.2

Let the result of chasing $Q$ with the dependencies $D$ be the query $U$. Let $M$ be a $D$-minimal query that is $D$-equivalent to $Q$. Recall that this means that we cannot remove any relational atoms from $M$, even if adding arbitrarily many equality atoms, without compromising $D$-equivalence to $Q$. We want to show that $M$ is isomorphic to a subquery of $U$.

Here are two classical results we use in the proof [AHV95]:

**Fact 1.** Any chase step of conjunctive query $Q$ with a dependency $d$ yields a query that is equivalent to $Q$ under $d$.

**Fact 2.** Suppose that the chase of conjunctive query $Q_1$ with the set of dependencies $D$ terminates, yielding query $U$. Then $Q_1$ is contained in conjunctive query $Q_2$ under $D$ if and only if there is a containment mapping from $Q_2$ into $U$.

Since we are working on a variation of conjunctive queries that allows explicit equality
atoms, we must redefine the notion of containment mapping:

**Definition B.1.1 (Containment Mapping)** Given conjunctive queries
\[
Q_1(x) \leftarrow B_1(x, y)
\]
and
\[
Q_2(z) \leftarrow B_2(z, u)
\]
where \(B_1, B_2\) are conjunctions of atoms that may be relational predicates or equality predicates and \(x, y, z, u\) are tuples of variables. A containment mapping from \(Q_1\) to \(Q_2\) is a function \(m\) from \(x, y\) to \(z, u\) such that (a) \(m(x) = m(z)\) is implied by the equality atoms in \(B_2\) and (b) for any atom \(A\) in \(B_1\), \(m(A)\) is implied by \(B_2\).

We also formalize the notion of subquery of a query:

**Definition B.1.2 (Subqueries)** A conjunctive query \(SQ\) is a subquery of a conjunctive query \(Q\) if there exists a containment mapping \(h\) from \(SQ\) into \(Q\) such that whenever the image of two distinct atoms \(R(x), R(y)\) under \(h\) coincides, the conjunction of equalities \(x = y\) is implied by the equality atoms in \(SQ\).

**Notation.** We use \(Q_1 \equiv_D Q_2\) to denote the \(D\)-equivalence of \(Q_1\) and \(Q_2\). \(Q_1 \subseteq_D Q_2\) means the containment of \(Q_1\) in \(Q_2\) over all instances satisfying \(D\), and \(Q_1 \subseteq Q_2\) means containment over all instances.

We will assume w.l.o.g. that for any two variables \(x, y\) of \(M\), whenever \(x = y\) is implied by \(D\) and the equalities in \(M\), it appears in \(M\)’s body. We say that the set of equalities in \(M\)’s body is transitively closed. This means that the C&B algorithm allows us to find all reformulations \(M\) with a minimal number of relational atoms, and the maximal number of equalities holding among their variables. \(^1\)

\(Q\) is contained in \(M\) under \(D\) and by fact 1, so is \(U\). By fact 2, there is a containment mapping \(h\) from \(M\) into \(U\). We now prove that

\(^1\)Of course, for any such \(M\), we can inspect all possible subsets of equalities, obtaining completeness for reformulations that are minimal in both relational and equality atoms. The minimality of the relational atoms is the important one however, since it implies minimal number of relational table scans.
(i) if variables $x, y$ have the same image under $h$, then $x = y$ is implied by the equalities in $M$.

(ii) if relational atoms $A_1, A_2$ have the same image under $h$, then $A_1 = A_2$ is implied by the equalities in $M$.

Notice that (i) and (ii) are equivalent to stating the isomorphism of $M$ to its image under $h$ (which is a subquery of $U$).

We prove (i) first. Assume the contrary and let $M'$ be obtained by adding the equality $x = y$ to $M$. Then $h$ is a containment mapping from $M'$ into $U$ according to definition 1. By facts 1 and 2, $Q =_D U \subseteq_D M'$. But $M' \subseteq M$ and by hypothesis $M \subseteq_D Q$. We conclude $M' =_D M$ which contradicts the transitive closure of the equality conditions in $M$.

We prove (ii) next. Suppose the contrary towards a contradiction. Let $S(m), S(n)$ be two relational atoms of $M$, who have the same image under $h$. Consider $M'$ obtained from $M$ by adding the conjunction of equality atoms $m = n$. Then $h$ is also a containment mapping from $M'$ into $U$, whence it follows by identical argument as in the proof of (i) that $M =_D M'$. But notice that $M'$ is equivalent over all databases with the query $M''$ obtained from $M'$ by eliminating $S(n)$. We have thus succeeded to preserve $D$-equivalence to $Q$ by eliminating a relational atom from $M$ and adding some equality atoms instead, thus contradicting $M$’s minimality. •

B.2 Proof of Proposition 4.2.3

We reduce the problem

CON given two conjunctive queries $Q_1, Q_2$,

decide whether $Q_1$ is contained in $Q_2$
on a class of models $C$ such that $Q_2$
returns a non-empty answer for at least
one model in $C$ (denoted $Q_1 \subseteq_C Q_2$).

to the problem
MIN  given conjunctive query $Q$ and set of embedded dependencies $D$,

decide if $Q$ is minimal under all models from $C$ that satisfy $D$.

The condition that $Q_2$ return a non-empty answer on at least some model $I \in C$ is easy to check in all common scenarios. When $C$ is the class of all $S$-instances, the canonical instance of $Q_2$ is an example for $I$. When $C$ is specified by a set of dependencies, the result of chasing $Q_2$ with these dependencies is an example, as long as it does not equate two constants, in which case $Q_2$ is equivalent to the empty query.

The reduction. The reduction is given in two steps, via the following auxiliary decision problem:

DISJ  given two conjunctive queries $P_1, P_2$,

decide whether $P_1 \subseteq_C P_2$ or $P_2 \subseteq_C P_1$.

Reducing CON to DISJ  Let

$$Q_1(x) \leftarrow body_1(x, y)$$

$$Q_2(x) \leftarrow body_2(x, z)$$

be conjunctive queries over schema $S$. Here $x$ denotes a tuple of variables $x_1, \ldots, x_n$, and similarly for $y, z$.

Let $e \notin S$ be a fresh, nullary predicate. For every $S$-instance $I$ from class $C$, there are two possible extensions to an $\{S \cup e\}$-instance $J$: one in which $e$ is interpreted as the empty set, and one in which it is interpreted as the singleton empty tuple. We denote with $C^e$ the class of models obtained by extending every model in $C$ in both ways.

Define

$$Q'_1(x) \leftarrow body_1(x, y), e()$$

We claim that

$$Q_1 \subseteq_C Q_2$$
\[ Q_1^e \subseteq C^e \quad Q_2 \vee Q_2 \subseteq C^e \quad Q_1^f \] (B.1)

**Proof:** Notice that since \( Q_2 \) is not defined in terms of \( e \), \( Q_2(J) = Q_2(I) \) regardless of \( e \)'s interpretation. Moreover, \( Q_1^e(J) = Q_1(I) \) when \( e \) is interpreted as non-empty, and \( Q_1^e(J) = \emptyset \) otherwise.

\[ \Rightarrow \] Pick an arbitrary \( J \in C^e \). If \( e \) is the empty set, \( Q_1^e(J) = \emptyset \subseteq Q_2(J) \). When \( e \) is interpreted as the singleton empty tuple, \( Q_1^e(J) = Q_1(I) \subseteq Q_2(I) = Q_2(J) \).

\[ \Leftarrow \] \( Q_2 \) returns a non-empty answer on at least one instance \( I \in C \). Then \( Q_2 \subseteq C^e \) \( Q_1^e \) is false. Indeed, \( Q_2 \) must have a non-empty answer also on the extension \( J \) of \( I \) with the empty set \( e \). But \( Q_1^e(J) = \emptyset \), which contradicts the containment statement.

It must therefore be the case that \( Q_1^e(J) \subseteq Q_2(J) \) is true for all \( J \in C^e \), in particular for those in which \( e \) is interpreted as non-empty, but on these, \( Q_1(I) = Q_1^e(J) \subseteq Q_2(J) = Q_2(I) \). Since the corresponding set of restrictions \( I \) of \( J \) is exactly \( C \), we have \( Q_1 \subseteq C \) \( Q_2 \).

**Reducing DISJ to MIN** Denote

\[
\begin{align*}
P_1(x) & \leftarrow \text{body}_1(x, y) \\
P_2(x) & \leftarrow \text{body}_2(x, z)
\end{align*}
\]

Let \( D \) be the set of dependencies \( \{c_1, b_1, c_2, b_2\} \) over the schema \( S' = S \cup \{P_1, P_2\} \), where \( P_1, P_2 \) are new \( n \)-ary relational symbols:

\[
\begin{align*}
(c_i) & \quad \forall x \forall y \ [\text{body}_i(x, y) \rightarrow P_i(x)] \\
(b_i) & \quad \forall x \ [P_i(x) \rightarrow \exists y \ \text{body}_i(x, y)]
\end{align*}
\]

Notice that, on any instance satisfying \( D \), the relation \( P_i \) contains exactly the result of the query \( P_i \). Also notice that \( c_i, b_i \) are exactly the kind of dependencies we use in the C&B approach to capture views.

We claim that

\[ P_1 \subseteq C \ P_2 \vee P_2 \subseteq C \ P_1 \]

\[ \Leftrightarrow \]
\[ P(x) \leftarrow P_1(x), P_2(x) \]

is not minimal over \( C \)-instances satisfying \( D \).

Notice that, on any instance satisfying \( D \), \( P \) defines the intersection of \( P_1 \) and \( P_2 \) when regarded as queries. \textit{Proof:} since \( P \) has only two atoms in its body, it is not minimal if and only if it is equivalent to either \( M_1(x) \leftarrow P_1(x) \) or \( M_2(x) \leftarrow P_2(x) \). But this is true if and only if when regarded as queries, \( P_1 =_C P_1 \cap P_2 \) or \( P_2 =_C P_1 \cap P_2 \), if and only if \( P_1 \subseteq_C P_2 \) or \( P_2 \subseteq_C P_1 \).

### B.3 Proof of Theorem 4.4.6

We repeat here the statement of theorem 4.4.6 from section 4.4.

\textbf{Theorem B.3.1} Let \( Q = \bigcup_{1 \leq i \leq M} Q_i \) be a \( D \)-satisfiable union of conjunctive queries such that the chase of each \( Q_i \) with the set of DEDs \( D \) terminates, yielding the \( m_i \) satisfiable leaves \( L_{i,1}, \ldots, L_{i,m_i} \). Denote \( U \) the union of conjunctive queries \( U = \bigcup_{1 \leq i \leq M, 1 \leq j \leq m_i} L_{i,j} \).

Then any \( D \)-minimal rewriting of \( Q \) under \( D \) is a subquery of \( U \).

\textit{Proof:} Let \( R = \bigcup_{1 \leq k \leq K} M_k \) be an arbitrary such minimal rewriting of \( Q \) under \( D \).

Assume w.l.o.g. that \( U \) contains no pairs of distinct queries \( L_{i,j}, L_{i',j'} \) such that \( L_{i,j} \) is contained in \( L_{i',j'} \) under \( D \) (if it does, throw them out, clearly preserving equivalence to \( U \)).

By definition of rewriting, \( M \) is equivalent to \( Q \) under \( D \), in particular \( Q \) is contained in \( R \) under \( D \), and by theorem 4.4.2 (1), so is \( U \). By theorem 4.4.2 (2), for every \( 1 \leq i \leq M, 1 \leq j \leq m_i \), either (i) the equality of two distinct constants is implied by the equalities in \( L_{i,j} \), or (ii) there is a \( 1 \leq q \leq K \) and a containment mapping from \( M_q \) into \( L_{i,j} \). Since all \( L_{i,j} \)'s are satisfiable, (i) cannot hold, so (ii) must.

Let \textit{targets}(\( M_q \)) denote the set of \( L_{i,j} \)'s into which there is a containment mapping from \( M_q \). This set is non-empty, otherwise we can safely remove \( M_q \) from \( R \) preserving equivalence, thus contradicting the minimality of \( R \). If \( M_q \) has no two atoms with the same relational symbol, any containment mapping into any \( L \in \text{targets}(M_q) \) satisfies the definition B.1.2 from section 4.1, so \( M_q \) is a subquery of \( L \).
Otherwise, we claim that there must exist an \( L \in \text{targets}(M_q) \) and a containment mapping \( h \) from \( M_q \) into it such that no two atoms of \( M_q \) have the same image under \( h \), which makes \( M_q \) a subquery of \( L \). Suppose the contrary towards a contradiction: Pick an arbitrary \( L \), a containment mapping into it, and let \( S(\overline{x}), S(\overline{y}) \) be two relational atoms of \( M_q \), who have the same image under \( h \): \( h(S(\overline{x})) = h(S(\overline{y})) \). Consider \( M_L \) obtained from \( M_q \) by adding the conjunction of equality atoms \( \overline{x} = \overline{y} \). Then \( h \) is also a containment mapping from \( M_L \) into \( L \), whence it follows from theorem 4.4.2 that \( L \) is contained in \( M_L \) over all instances (under the empty set of dependencies), and in particular under \( D \). But at the same time \( M_L \) is contained in \( M_q \) on all instances (we added equality conditions to the latter to obtain the former, making it more restrictive). Summarizing, we have

\[
U = \bigcup_{1 \leq i \leq M, 1 \leq j \leq m_i} L_{i,j} \subseteq_D \bigcup_{1 \leq k \leq K, k \neq q} M_k \cup \bigcup_{L \in \text{targets}(M_q)} M_L \subseteq_D \bigcup_{1 \leq k \leq K} M_k = R =_D U
\]

in other words, by replacing \( M_q \) in \( R \) with \( \bigcup_{L \in \text{targets}(M_q)} \) we preserve equivalence under \( D \). This contradicts the minimality of \( R \). and \( M_q \) We have just proven the equivalence of \( M_q' \) with \( L_{i,j} \) under \( D \).

It follows that every \( M_q \) is a subquery of some \( L_{i,j} \), which is what we had to prove.

End of proof.
Appendix C

The Theory of Chase-Based Reformulation for XQueries

We first introduce some notions that will be used in the proofs.

C.1 Applying the C&B Method to XQuery Reformulation

We introduce some terminology first.

**MARS- and TIX-instances.** A MARS instance is a relational instance of schema $R$, enhanced with a collection of XML documents. Internally, we view every MARS instance as a relational instance over the combined schemas $R$ and TIX, by representing the XML documents in the generalized relational encoding described in section 3.1. This relational instance satisfies a certain set of constraints and is called a $(TIX,R)$-instance.

Recall that the encoding involves a child relation which will model a forest, containing one tagged tree for each XML document. For every XML document, there will be a unary predicate holding a single node identity, namely that of the root. Moreover, $el$ is the set of identities of the nodes in the XML documents and the desc relation is the reflexive, transitive closure of the child relation.

**Intended TIX-instances.** Note that while any MARS instance corresponds to some $(TIX,R)$-instance, the converse is not true: there are infinitely many $(TIX,R)$-instances where the child relation does not correspond to a forest, or $el$ and desc are proper...
supersets of the intended interpretations (el may contain unreachable nodes, and desc may contain pairs of nodes such that the second component is not a descendant of the first). We shall call the (TIX, R)-instances corresponding to MARS instances intended instances. They are characterized as follows:

(i) the child relation corresponds to a forest

(ii) el consists precisely of the endpoints of child edges and the targets of references.

(iii) desc(e, e′) belongs to the instance if and only if the element node of identity e′ is reachable from that of identity e along a chain of child edges. Equivalently, we say that given the child relation, desc is the minimal relation satisfying the (refl), (base), (trans) constraints.

Reducing containment over MARS-instances to containment over (TIX, R) instances. Clearly, the equivalence (and more generally containment) of two XBind queries, BQ₁, BQ₂ holds on all MARS instances if and only if it holds for T(BQ₁), T(BQ₂) on all intended (TIX, R)-instances. However, what the C&B method offers us is a decision procedure for containment of T(BQ₁), T(BQ₂) an all (TIX, R)-instances. While this equivalence is sufficient for that of BQ₁, BQ₂ on MARS instances, it is a priori not clear that is is also necessary.

The difficulty stems from the fact that we have no way of ruling unintended instances out using first-order logic statements (therefore DEDs), because the transitive closure and treeness properties are notoriously inexpressible in first-order logic [EF95].

In theorem C.3.1 we show nevertheless that for a significant class of XQueries, schema mappings and XML integrity constraints, the containment of T(BQ₁), T(BQ₂) on arbitrary TIX-instances is equivalent to their containment on intended TIX-instances, and therefore to the containment of BQ₁, BQ₂ on MARS instances. A corollary of this result is that the minimal reformulations of an XBind query BQ are in one-to-one correspondence to the minimal rewritings of T(BQ) and therefore our C&B-based strategy for reformulation is complete.

 Unsatisfiability. The following issue sheds additional light on the difference between deciding properties of queries over all instances, as opposed to over intended instances
only. In the absence of constants in the language, given arbitrary conjunctive query \( q \), there is always some relational database on which \( q \) returns a non-empty answer. We say that \( q \) is *satisfiable*. If constants are present in the language, unsatisfiability can occur from equality tests between distinct constants (recall how we dealt with this in theorem 4.4.2). But even when no constants are allowed, it is not true that \( q \) is satisfiable by an *intended* instance:

**Examples.** Let \( q \) be the conjunctive query obtained by compiling the following XQuery binding part:

\[
\text{for } x \text{ in } //a, y \text{ in } x/b, z \text{ in } x/c, \text{ where } y == z
\]

\( q \) is unsatisfiable over intended instance (it returns the empty answer on all of them) because \( y, z \) are bound to \( b \)-respectively \( c \)-elements, which therefore can never be identical. The same holds if we replace the equality test \( y == z \) with \( y = z \), thus testing for value-equality, that is isomorphism: \( b \)-elements and \( c \)-elements cannot be isomorphic because they do not have the same tag.

Similarly,

\[
\text{for } x \text{ in } //a, y \text{ in } x/b, z \text{ in } y/c, \text{ where } x == z
\]

is unsatisfiable because it demands that \( y \) be simultaneously a proper descendant and a proper ancestor of \( x \) (here == tests for node identity).

Less obviously, an XBind query may be satisfiable on intended instances, but not on those satisfying certain constraints. Let \( q \) correspond to the following XQuery binding part (it has no *where* clause):

\[
\text{for } x \text{ in document(''foo.xml'')/a, y in } x/b
\]

\( q \) is certainly satisfiable by an intended instance, as it has a non-empty match on the document \(<a><b>foo</b></a>\).

Now consider the constraint
saying that all b-elements are immediate subelements of some c-element which is a child of the root. Obviously, q is unsatisfiable on all documents that satisfy (constr). Contrast again with relational conjunctive queries which are known to be satisfiable under any set of relational embedded dependencies. ●

**C-unsatisfiability.** We say that an XBind query q is C-unsatisfiable if it returns the empty answer on all MARS instances satisfying the set of constraints C. Equivalently, q is C-unsatisfiable if $T(q)$ returns the empty answer on all intended (TIX, R)-instances. If $C = \emptyset$, we call q unsatisfiable.

A complete solution for deciding containment needs to detect unsatisfiable XBind queries, because they are vacuously contained in any XBind queries. More importantly, when trying to reformulate an XQuery, it is desirable to detect its unsatisfiability and report an error message to the user. Fortunately, it turns out that we can do this for a significant class of constraints (see proposition C.2.1 in section C.2).

**Canonical instance and canonical mapping.** We use the idea of canonical instance introduced in [CM77]: given a conjunctive query Q, its associated canonical instance $CInst(Q)$ has a relation R for every relational symbol appearing in Q’s body. Q’s variables are partitioned into equivalence classes by the equality atoms in the body. For every equivalence class $[x]$, $CInst(Q)$ contains a constant $c_x$. For every relational atom $R(x_1, \ldots, x_n)$ of Q, $CInst(Q)$ contains the R-tuple $(c_{x_1}, \ldots, c_{x_n})$. We will call the mapping $\{x \mapsto c_x\}$ for every variable x the canonical mapping from Q into $CInst(Q)$. Note that the canonical mapping is a valuation and therefore, denoting with $y_1, \ldots, y_m$ the head variables of Q, we have that $(c_{y_1}, \ldots, c_{y_m}) \in Q(CInst(Q))$, that is the image of Q’s head tuple under the canonical mapping belongs to the result of evaluating Q on its own canonical instance.

**Supported desc atoms.** Since the child and desc relational symbols are binary, we will sometimes refer to them as child- and desc-edges in the canonical instance. We say that an atom desc$(x, y)$ in Q is supported if and only if $c_y$ is reachable in $CInst(Q)$ from $c_x$ along a path of child- and desc-edges. This path may be empty when $x = y$
is implied by the conditions in $Q$, because then $[x] = [y]$ so $c_x$ and $c_y$ are identical. The corresponding desc edge in $CInst(Q)$ is called supported as well. For example, in $q(z) \leftarrow child(x,y), desc(y,z), desc(x,y), desc(x,z)$, we have that $desc(x,y)$ and $desc(x,z)$ are supported, while $desc(y,z)$ is not.

**t-supported instance.** Given conjunctive query $Q$ and arbitrary string $t$, we say that $I_s$ is the $t$-supported instance of $Q$, denoted $I_s = CInst_s^t(Q)$, if and only if $I_s$ is obtained from $Q$’s canonical instance, $CInst(Q)$ by adding the pair of tuples $child(c_x,c_y), tag(c_y,t)$ for every unsupported atom $desc(x,y)$ in $Q$. Exemplifying on $q$ above, $CInst_s^{V''}(q) = \{child(c_x,c_y), desc(c_y,c_z), desc(c_x,c_y), child(c_y,c_z), tag(c_z,'''V''')\}$. Note that this makes all desc atoms in $I_s$ supported, whence the name.

## C.2 Unsatisﬁability

In section C.1 we pointed out a major difference between containment of binding queries over arbitrary instances, versus their containment over intended ones. In the absence of constants in the language, given arbitrary conjunctive query $q$, there is always some relational database on which $q$ returns a non-empty answer, but nothing guarantees the existence of an intended instance. If constants are present in the language, unsatisﬁability can occur from equality tests between distinct constants even on non-intended instances.

**C-unsatisﬁability.** We say that an XBind query $q$ is C-unsatisﬁable if it returns the empty answer on all MARS instances satisfying the set of constraints $C$. Equivalently, $q$ is C-unsatisﬁable if $T(q)$ returns the empty answer on all intended $TIX,R$-instances satisfying $C$. If $C = \emptyset$, we call $q$ unsatisﬁable.

We need to detect unsatisﬁable XBind queries, because they are vacuously contained in any XBind queries. It turns out that we can do this for a restricted class of constraints:

**Well-behaved DEDs.** A DED is well-behaved if whenever the premise of the implication contains an atom $child(x,y)$, it also contains an atom $tag(y,"t")$ for some string constant "t". For example, (constr) above is well-behaved, and so are all constraints shown in Chapter 5, but (base) from TIX is not.

**Proposition C.2.1** Let $BQ$ be an XBind query, whose disjunctive normal form is the union of disjunction-free queries $\bigcup_{1 \leq i \leq M} Q_i$. Let $C$ be a set of well-behaved constraints
such that for each $1 \leq i \leq M$ the chase of $T(Q_i)$ with $C \cup TIX$ terminates, yielding the leaves $L_{i1}^1, \ldots, L_{im_i}^i$. Then $BQ$ is $C$-unsatisfiable if and only if for every $1 \leq i \leq M$ and $1 \leq j \leq m_i$:

1. there is a pair of distinct constants $c_1, c_2$ whose equality is implied by the conditions in $L_j^i$, or

2. $L_j^i$ contains an atom $\text{child}(x, y)$ where $x = y$ is implied by the conditions in $L_j^i$.

Proof: “if”: From theorem 4.4.2, it follows that $T(Q_i)$ is equivalent to $\bigcup_j L_j^i$ on all instances satisfying $TIX \cup C$, and in particular on all intended $(TIX, R)$-instances. But each $L_j^i$ is unsatisfiable. In case (1), the unsatisfiability of $L_j^i$ holds for arbitrary instances. In case (2), it holds for intended instances, because their $\text{child}$ relation is a forest, thus ruling out valuations from atoms as in (2).

“only if”: We show equivalently that if some $L_j^i$ violates (1) and (2), then it is satisfiable. We show that the intended instance witnessing the satisfiability is the $t$-supported canonical instance $I$ of $L_j^i$:

Lemma C.2.2 For every $i, j$, if (1) and (2) do not hold for $L_j^i$, then for any string $t$ that does not appear in $BQ_1$ or $C$, $I = C\text{Inst}^t(t)(L_j^i)$ is an intended $(TIX, R)$-instance satisfying $C$.

Given this result, observe that the canonical mapping $\{x \mapsto c_x\}$ is a valuation from $L_j^i$ into $I$, so the former’s answer is nonempty, i.e. $L_j^i$ is satisfiable.

Proof of lemma C.2.2: Denote $I = C\text{Inst}^t(t)(L_j^i)$. According to the definition of intended instances (page 161), we must show that

(i) $I$’s $\text{child}$ relation corresponds to a forest,

(ii) $e_1$ contains precisely the endpoints of $\text{child}$ edges and targets of references

(iii) $\text{desc}(e, e')$ belongs to the instance if and only if the element of identity $e'$ is reachable from that of identity $e$ along a chain of $\text{child}$ edges.

In addition, we must show that (iv) $I$ satisfies the constraints in $C$.

By definition of the chase, we know that no chase steps with constraints from $TIX$ or $C$ are applicable to $L_j^i$. It follows from the definition of constraint satisfaction (section 4.4.2)
that $CInst(L^i_j)$ satisfies all of these constraints. We show in claim C.2.3 below that the same holds for $I$ (the proof will be shown shortly).

**Claim C.2.3** Suppose that $L^i_j$ has $n$ unsupported edges $\text{desc}(u_k, v_k)$ ($1 \leq k \leq n$), and define $A_0 = L^i_j$, and let $A_{k+1}$ be the query obtained from $A_k$ by adding to it the atoms $\text{child}(u_{k+1}, v_{k+1}), \text{tag}(v_{k+1}, "t")$. In particular, $A_n = (L^i_j)^*$. Then for any $1 \leq k \leq n$, no chase step with constraints from $C \cup TIX$ applies to $A_k$ (or equivalently, each $CInst(A_k)$ -and $I$ in particular- satisfies all constraints).

This immediately implies (iv).

(ii) follows because the $\text{el}$ tuples in $I$ coincide with those $CInst(L^i_j)$, and because no $\text{el}$ atom appears in $Q_i$ to begin with, therefore it is brought into $L^i_j$ by the chase. An easy induction on the length of the chase sequence proves that the only $\text{el}(x)$ atoms added correspond to the case when $x$ is an endpoint of a $\text{child}$ edge ($\text{el}_c$), a reference target ($\text{el}_t$) or, when none of the above holds, an endpoint of an unsupported $\text{desc}$ edge ($\text{el}_d$) in $CInst(L^i_j)$. But the latter’s endpoints are also the endpoints of a $\text{child}$ edge added by construction to $I$.

(iii) By construction of $I$, all $\text{desc}$ edges are supported, whence it follows easily that for every tuple $\text{desc}(x, y)$, $y$ is reachable in $I$ from $x$ along a chain of $\text{child}$ edges. The converse is proved by induction on the length of the chain from $x$ to $y$, using the fact that if there are $\text{child}$ edges in $I$ which do not appear in $CInst(L^i_j)$, these correspond to (unsupported) $\text{desc}$ edges in $CInst(L^i_j)$. We also use the inapplicability of the ($\text{base}$) and ($\text{trans}$) constraints.

(i) follows because the satisfaction of ($\text{noLoop}$) implies that there is no non-trivial cycle of $\text{child}$ edges in $I$, while the assumption $\neg$(2) rules out trivial cycles. This makes $I$’s $\text{child}$ relation correspond to a directed acyclic graph. By the satisfaction of ($\text{oneParent}$), no element has two parents, so the $\text{child}$ relation is really a forest. The forest is tagged: the satisfaction of ($\text{someTag}$) ensures that each element has at least one tag. The satisfaction of ($\text{oneTag}$) ensures that every element has at most one tag, or else $L^i_j$ would contain an equality between two tag constants of the same element, thus violating the assumption $\neg$(1).

**Proof of claim C.2.3** By induction on $k$. 

167
Base: $A_0$ coincides with $L^i_j$ which is a leaf in the chase tree, so no further chase steps apply to it.

Step: Assume that no chase step applies to $A_k$. Consider the unsupported atom $\text{desc}(u_{k+1}, v_{k+1})$ and denote it $\text{desc}(x, y)$ for brevity. Assume that the corresponding atoms $\text{child}(x, y), \text{tag}(y, " t'')$ in trigger a chase step of $A_{k+1}$ with some constraint $c$. $c$ cannot belong to $C$, because recall that the constraints in $C$ are well-behaved, so each of the $\text{child}$ atoms in the premise of their implication is accompanied by a $\text{tag}$ atom whose second component is distinct from the constant “$t$” by construction of $I$. Therefore there is no homomorphic mapping from any constraint in $C$ to $I$ using $\text{child}(x, y)$ or $\text{tag}(y, " t'')$. The only constraints in TIX that might map into $\text{child}(x, y)$ or $\text{tag}(y, " t'')$ are $(\text{oneParent}), (\text{choice})$ and $(\text{el}_c)$ because they contain $\text{child}$ atoms without corresponding $\text{tag}$ atoms. We show that if a chase step with any of these applies, it follows that $\text{desc}(x, y)$ was supported in $L^i_j$ to begin with, which is a contradiction.

- Assume $c = (\text{oneParent})$. If $\text{child}(x, y)$ triggers a chase step with $c$, there must exist an atom $\text{child}(z, y)$ in $A_{k+1}$ such that $z = x$ is not implied by its conditions. The same must then hold for $A_k$. By the inapplicability of $(\text{base})$ to $A_k$, the latter must also contain a $\text{desc}(z, y)$ atom. But then there is a homomorphism from the premise of $(\text{inLine})$ into $A_k$, with image $\text{desc}(x, y), \text{desc}(z, y)$. Since $(\text{inLine})$ is inapplicable, its conclusion must be implied by the conditions in $A_k$. That is, either $z = x$ or $\text{desc}(z, x)$ or $\text{desc}(x, z)$ are implied in $A_k$ and hence in $A_{k+1}$. The case $z = x$ is ruled out by the applicability of the chase step with $c$ to $A_{k+1}$. In the case $\text{desc}(x, z)$, the presence of atoms $\text{desc}(x, z), \text{desc}(z, y)$ in makes $\text{desc}(x, y)$ supported in $A_k$ and by construction, supported in $L^i_j$ to begin with. Contradiction. Finally, in case $\text{desc}(z, x)$ is implied, there is a homomorphism from the premise of $(\text{choice})$ into $A_k$’s atoms $\text{child}(z, y), \text{desc}(z, x), \text{desc}(x, y)$. By the inapplicability of $(\text{choice})$ to $A_k$, either $z = x$ or $y = x$ is implied in $A_k$. But each case makes $\text{desc}(x, y)$ supported in $A_k$, therefore in $L^i_j$, contradicting our premise.

- Assume $c = (\text{choice})$. Then $A_{k+1}$ must contain atoms $\text{desc}(x, z), \text{desc}(z, y)$, and therefore so must $A_k$, whence we obtain the contradiction that $\text{desc}(x, y)$ is supported in $A_k$, and therefore in $L^i_j$.

- Assume $c = (\text{el}_C)$. If $\text{child}(x, y)$ triggers a chase step with $c$, then $A_{k+1}$ cannot contain both $\text{el}(x)$ and $\text{el}(y)$ atoms. But then the same holds of $A_k$, which must also
contain the (unsupported) desc\((x,y)\) atom. It follows that a chase step of \(A_k\) with (e1_D) applies, thus contradicting the induction hypothesis. **End of proof of claim C.2.3.**

**End of proof of lemma C.2.2.**

**End of proof of proposition C.2.1.**

Obviously, if XBind query \(BQ_1\) is unsatisfiable, it is trivially contained in any other XBind query. Recall that our motivation for checking containment is that of reformulating \(BQ_1\) equivalently. In case it turns out to be unsatisfiable, we can shortcut stage 2 of our reformulation algorithm (the backchase). In the next section, we give a characterization of non-trivial containment.

### C.3 A Characterization of Containment over Intended Instances

**Well-behaved XQueries.** We can decide containment for an important fragment of XQueries, which we call *well-behaved* (defined in section A.1). The most notable operations disallowed are the range predicates, arbitrary negation, universal quantification, FILTER and user-defined functions, aggregates.

Also ruled out are navigation to a child of unspecified tag (\(\ast\) in abbreviated XPath notation) or to a parent (\(\cdots\)).

**Theorem C.3.1** Let \(BQ_1, BQ_2\) be XBind queries of behaved XQueries, whose disjunctive normal forms are the unions of disjunction-free queries \(\bigcup_{1 \leq i \leq M} Q_i\) respectively \(\bigcup_{1 \leq k \leq N} P_k\). Let \(C\) be a set of well-behaved constraints and assume that for each \(1 \leq i \leq M\) the chase of \(T(Q_i)\) with \(C \cup TIX\) terminates. Denote the set of \(C\)-satisfiable leaves of the chase tree with \((L^i_j)_{1 \leq j \leq m_i}\) (\(m_i = 0\) if there are none). Then \(BQ_1 \subseteq_C BQ_2\) if and only if for every \(1 \leq i \leq M\) and every \(1 \leq j \leq m_i\) there is some \(1 \leq k \leq N\) and a containment mapping from \(T(P_k)\) into \(L^i_j\).

**Proof:** Note that if \(BQ_1\) is \(C\)-unsatisfiable, there are no containment mappings to check and the containment holds vacuously. We therefore only need to prove the result for satisfiable \(BQ_1\).
“if”: Let \( t \) be a tuple in the result of \( \bigcup_i T(Q_i) \) over an arbitrary instance \( I \). Since \( BQ_1 \) is satisfiable, there must be an \( i \) and a valuation \( v \) from \( T(Q_i) \) to \( I \) such that the image of \( Q_i \)'s head tuple under \( v \) is \( t \). But by hypothesis, there is a \( k \) and a containment mapping \( m \) from \( T(P_k) \) into \( T(Q_i) \). It is easy to see that \( v \circ m \) is a valuation from \( T(P_k) \) into \( I \), such that the image of \( P_k \)'s head is \( t \). Therefore \( t \) belongs to the answer of \( \bigcup_{k} T(P_k) \). Since \( t \) and \( I \) were picked arbitrarily, this entails that the containment holds on all arbitrary TIX-instances, and in particular on all intended ones. But containment on all intended instances is equivalent to the containment \( BQ_1 \subseteq C \subseteq BQ_2 \).

“only if”: Note first that \( BQ_1 \subseteq C \subseteq BQ_2 \).

is equivalent to

\[
\bigcup_{1 \leq i \leq M} Q_i \subseteq C \subseteq \bigcup_{1 \leq k \leq N} P_k
\]
on all MARS instances and therefore to

\[
\bigcup_{1 \leq i \leq M} T(Q_i) \subseteq C \subseteq \bigcup_{1 \leq k \leq N} T(P_k)
\]
on all intended \((TIX, R)\)-instances. But by theorem 4.4.2, \( T(Q_i) \) is equivalent to \( \bigcup_{1 \leq j \leq m_i} L_i^j \) on all \((TIX, R)\)-instances satisfying \( C \), and in particular on all intended such instances. We therefore have that

\[
BQ_1 \subseteq C \subseteq BQ_2
\]

\[
\Leftrightarrow
\]

\[
\text{intended \((TIX, R)\)-instances satisfying } C \mid= \bigcup_{1 \leq i \leq M, 1 \leq j \leq m_i} L_i^j \subseteq \bigcup_{1 \leq k \leq N} T(P_k)
\]

Now fix arbitrary \( i', j' \) and let \( I \) be the \( t \)-supported instance of \( L_i^{j'} \): \( I = CInst_s(L_i^{j'}), \)
where \( t \) does not appear in any \( L_q^p \) (for all \( p, q \)) or in \( T(P_k) \) (for all \( k \)). Denote with \( ht \) the head tuple of \( BQ_1 \) (which is also that of each \( T(Q_i) \)), and with \( cm \) the canonical mapping corresponding to \( L_i^{j'} \). Then \( cm(ht) \in L_i^{j'}(I) \). To see why, recall that \( cm(ht) \in L_i^{j'}(CInst(L_i^{j'})) \) and that \( I \) contains \( CInst(L_i^{j'}) \) by construction, so \( cm \) is at the same time a valuation from \( L_i^{j'} \) into \( I \). It follows that \( cm(ht) \in (\bigcup_{1 \leq i \leq M, 1 \leq j \leq m_i} L_i^j)(I) \). But by lemma C.2.2 and equivalence (C.1) it follows that \( cm(ht) \in (\bigcup_{1 \leq k \leq N} T(P_k))(I) \) and from
here that there is a $k'$ and a valuation $v$ from $T(P_{k'})$ into $I$ such that the image of $T(P_{k'})$'s head tuple under $v$ is $cm(ht)$. The following claim concludes the proof:

**Claim C.3.2** The image of $T(P_{k'})$ under $v$ is contained in $CInst(L_{j'}^u)$.

Given this result, we have that $cm^{-1} \circ v$ is a homomorphism from $T(P_{k'})$ into $L_{j'}^u$ which maps $T(P_{k'})$'s head tuple into $ht$, or in other words $v$ is a containment mapping, which is what we had to prove.

**Proof of claim C.3.2:** We show that none of the child and tag tuples added to $CInst(L_{j'}^u)$ in order to obtain $I$ can be in the image of $T(P_{k'})$ under any valuation, hence under $v$. Assume toward a contradiction that some conjunct child$(x;y)$ from $T(P_{k'})$'s body maps to a tuple child$(c_u,c_w)$ from $I$, which was added because of the unsupported conjunct desc$(u,w)$ in $L_{j'}^u$. By construction of $I$, it must also contain a tuple tag$(c_w,"t")$. But $BQ_1$ is well-behaved. An inspection of the syntax for well-behaved XPaths as well as of the translation operator $T()$, together with the observation that desc$(u,w)$ is unsupported reveal that tag$(c_w,"t")$ is the only tag tuple in $I$ involving $c_w$.

$BQ_2$ is well-behaved too, which means (among others) that for all XPath expressions appearing in it, every child navigation step specifies some name $n$ of the target subelement. Then the conjunct child$(x,y)$ in $T(P_{k'})$ was obtained by the translation of such a navigation step, so $T(P_{k'})$ must also contain a tuple tag$(y,"n")$, which must map to the unique tag$(c_w,"t")$ in any mapping that maps $y$ to $c_w$. But by construction of $I$, $t$ was chosen distinct from $n$, therefore the map is not a valuation, which is the contradiction we were seeking. **End of proof of claim C.3.2. End of proof of theorem C.3.1.**

## C.4 A Sufficient Condition for the Termination of the Chase

Observe that theorem C.3.1 gives a decision procedure for containment only if the chase with the constraints in $C \cup TIX$ terminates. In this section, we introduce a sufficient condition for the termination of the chase and show that it is satisfied when the constraints are obtained as a result of compiling schema mappings as shown in Chapter 5.

**Proposition C.4.1** Assume that the schema mapping is given by a set of well-behaved XQueries and default encodings of relations as XML (as discussed in Chapter 5). Let $\Sigma_{Map}$
be the set of DEDs to which the schema mapping compiles. Assume also that the integrity constraints on the relational storage are full (i.e. they have no existential quantifiers) and the XML integrity constraints are bounded. Let $\Sigma_{IC}$ be the set of DEDs to which all integrity constraints compile. Then

(1) Given behavedXBIND query $BQ$, the chase of $T(BQ)$ with $TIX \cup \Sigma_{Map} \cup \Sigma_{IC}$ terminates.

(2) Moreover, if the schema mapping corresponds to a pure LAV or GAV approach (the storage schema is defined as view of the published schema, respectively the other way around), the chase terminates in time polynomial in the size of $T(BQ)$, and exponential in the maximum size of a constraint in $\Sigma_{Map} \cup \Sigma_{IC}$.

Note that for a given configuration of the reformulation algorithm, $\Sigma_{Map} \cup \Sigma_{IC}$ are fixed (given as parameters) so the chase terminates in time polynomial in the size of the input (the XBIND query). Of course, the ensuing backchase minimization phase will be exponential in the size of the query.

Proof sketch: The proof uses several observations.

The first is that in order for the chase to diverge, it must be the case that an infinite number of fresh variables are introduced during the chase. Otherwise, there is only a finite number of relational atoms one can build over a finite set of variables using a finite schema, and once all of them are introduced, no more chase step applies.

The second is that the only DEDs containing existential quantifiers (and therefore generating fresh variables during the chase) are those describing the XML integrity constraints, or the binding, Skolem and copy relations (recall pages 30, 45, and 53, respectively) used to describe the semantics of XQueries and of relational-to-XML encodings. We will address XML integrity constraints later. Since the binding, Skolem and copy relations are local to each XQuery/encoding $Q$, invisible in the published or storage schema, they cannot be mentioned by other XQueries/encodings/integrity constraints. The only way a fresh variable introduced during the chase with these constraints will contribute to the chase with constraints not pertaining to the compilation of $Q$ is if these variables propagate to atoms belonging to the visible published and virtual schema schema.
A third observation is that while two X Queries may share the same source schema, each of them has its own, separate output schema. Therefore, atoms are added to an output schema only by its corresponding X Query, and the graph showing the flow of variables between atoms from the various parts of the visible (i.e. published and storage) schema is acyclic (actually a forest). It follows that if the chase diverges, there must exist an infinite chase sequence using only the constraints stemming from

(i) the compilation of the same X Query/encoding, or
(ii) the compilation of XML integrity constraints over a single document, or
(iii) the integrity constraints on the relational storage.

But case (iii) is excluded, because these integrity constraints have no existential quantifiers and therefore introduce no fresh variables at all, so the first observation applies.

Finally, a detailed study of the constraints generated from a given X Query shows that divergence is ruled out, which eliminates case (i). Similarly, as shown in section 9.2, the chase with bounded SXICs terminates, which rules out case (ii). End of proof sketch.

C.5 Proof Sketch for Theorem 6.2.1 (Relative Completeness)

The proof parallels that of Theorem 4.4.6 (shown in section B.1). The only change is that Fact 2 is replaced by Theorem C.3.1, which was stated and proven in section C.3.
Appendix D

More on XPath Containment
(Details for Chapter 9)

D.1 Detailed Treatment for Simple XPath

D.1.1 Upper Bounds

We first point out a major difference between containment for simple XPaths and SXICs versus containment for conjunctive queries and DEDs. In the absence of constants in the language, given arbitrary conjunctive query $p$ and set of DEDs $D$, there is always a relational database satisfying $D$, on which $p$ returns a non-empty answer. We say that $p$ is $D$-satisfiable. This is not always the case if $p$ is a simple XPath expression and $D$ a set of SXICs. As an example, let $p$ be $/\text{person}@\text{ssn} = ./\text{loves}/\text{person}@\text{ssn}$ returning persons who love persons of same social security number (in particular, $p$ may return narcissistically inclined individuals). Let $D$ contain the key constraint $(\text{key}_{s,p})$ on the $\text{ssn}$-attribute of $\text{person}$-elements (shown in section 9.1). $p$ is $D$-unsatisfiable (it returns the empty answer on all documents satisfying $D$) because any XML document satisfying the key constraint cannot nest a $\text{person}$-element $e$ in a $\text{person}$-element that agrees with $e$ on the $\text{ssn}$-attribute: this would amount to nesting $e$ within itself. We need to detect $D$-unsatisfiable XPaths, because they are vacuously contained in any other XPath under $D$. If constants are present in the language, unsatisfiability can occur even in the relational case, from equality tests between distinct constants. Our decision procedure is given in
Theorem D.1.1 Given simple XPaths $p_i$ ($i = 1, 2$) and the set $C$ of bounded SXICs, let $p_i$ be translated to the union of conjunctive queries $Q^1_i, \ldots, Q^n_i$, and let $\Sigma_C$ denote the result of $C$’s translation to a set of DEDs according to $T()$. Then the following are true

- The chase of $Q^1_j$ with $TIX \cup \Sigma_C$ terminates for every $1 \leq j \leq n_1$ and, the depth of the chase tree is polynomial in the size of $Q^1_j$ and exponential in that of the constraints in $\Sigma_C$. Denote the leaves with $\{L_1, \ldots, L_m\} = \bigcup_{j=1}^{n_1} \text{chase}\, TIX \cup \Sigma_C(Q^1_j)$.

- $p_1$ is contained in $p_2$ under $C$ if and only if for every $1 \leq i \leq m$ either
  1. there is a homomorphism from the formula $\text{child}(x', y') \wedge \text{desc}(y', x')$ into $L_i$, or
  2. the equality of distinct string constants $s_1, s_2$ is implied by the equalities in $L_i$, or
  3. there is a $1 \leq j \leq n_2$ and a containment mapping from $Q^2_j$ into $L_i$.

The theorem follows as a special case of theorem C.3.1, observing that any simple XPath can be translated to the binding part of a well-behaved XQuery.

We give the intuition behind the conditions (1),(2),(3) above. (1) detects queries $L_i$ which test the existence of a nontrivial cycle in the XML document, thus being unsatisfiable. (1) is obviously PTIME-checkable. (2) detects queries that contain unsatisfiable tests (they could result from testing for elements with two distinct tags, or with two non-IDREFS attributes of same name but distinct values). It can be checked in PTIME by checking the membership of $(s_1, s_2)$ in the symmetric, reflexive, transitive closure of the equality conditions of $L_i$. There are conceivably other reasons for $L_i$’s unsatisfiability (e.g. a test for two distinct paths leading to the same node). It turns out however that, no matter what the reasons are, one of the conditions (1) or (2) must apply, as a result of chasing with the DEDs $(\text{noLoop}), (\text{oneParent}), (\text{noShare}), (\text{inLine})$ from TIX. Therefore, if none of (1),(2) applies, $L_i$ is satisfiable and (3) turns out to be equivalent to containment in $Q^2_j$.

Example: Containment by condition (1). The simple XPath expression $p$ above is shown to be $\{(\text{key}s,p)\}$-unsatisfiable as follows. Let $T(p) = p'$ where

$$
p'(y) \leftarrow \text{desc}(\text{root}, x), \text{child}(x, y), \text{tag}(y, \text{person}), \text{attr}(y, \text{ssn}, z),
\text{child}(y, u), \text{tag}(u, \text{loves}), \text{child}(u, v), \text{tag}(v, \text{person}), \text{attr}(v, \text{ssn}, z)
$$

175
By chasing $p'$ in order with $(\text{key}), (\text{oneParent}), (\text{base})$, we obtain a query $p''$ that extends $p'$ with the atoms $y = v, x = u, \text{desc}(u, v)$ respectively. Note that condition (1) applies now, as witnessed by the homomorphism $h = \{x' \mapsto y, y' \mapsto u\}$. The chase continues since more steps are applicable, but they cannot affect the existence of $h$, as they only add atoms to $p''$. 

Example: Containment by condition (3). We highlight here how we deal with the $// \text{ operator. Given } q_1 = /A/B \text{ and } q_2 = //B//., \text{ it is easy to see that } q_1 \text{ is contained in } q_2 \text{ over all XML documents (i.e. even if } C = \emptyset). \text{ We show how we infer this using condition (3). The translation yields } q'_1(x) \leftarrow \text{child(root, } x_1), \text{child}(x_1, x), \text{tag}(x, B) \text{ and } q'_2(y) \leftarrow \text{desc(root, } y_1), \text{child}(y_1, y_2), \text{tag}(y_2, B), \text{desc}(y_2, y). \text{ Note that there is no containment mapping from } q'_2 \text{ to } q'_1 \text{ as the latter contains no desc-atoms to serve as image for the former’s desc-atoms. But by chasing } q'_1 \text{ with } (\text{base}), (\text{el}), (\text{refl}) \text{ we add } \text{desc(root, } x_1), \text{el}(x_1), \text{el}(x), \text{desc}(x, x) \text{ to } q'_1, \text{ thus creating an image for the containment mapping } \{y \mapsto x, y_1 \mapsto x_1, y_2 \mapsto x\}. \text{ There are further applicable chase steps, omitted here as they only add new atoms and hence do not affect the existence of the containment mapping.} 

The upper bounds for containment given in theorem 9.2.1 follow as a corollary of theorem D.1.1.

Proof of Theorem 9.2.1: (1) We prove equivalently that non-containment is in $\Sigma^P_{2}$, that is it is decidable by an NP machine with an NP oracle. In the notation of theorem D.1.1, the machine guesses $Q_i$, then the root-leaf path in the chase tree of $Q_i$ leading to some $L_j$ as follows. The necessary space is polynomial in the size of $p_1$ and the maximum size of a DED in $\Sigma_C$: for every step of the root-leaf path in the chase tree, the machine guesses the DED $d$ that applies, the homomorphism $h$ from $d$’s left-hand side of the implication, and the disjunct (in $d$’s right-hand side of the implication) which is used to chase on this particular path. This information is sufficient to check in PTIME (in the size of $d$) whether the guessed step corresponds to a chase step. Then the machine uses the oracle to check that this chase step is indeed applicable (it must ask whether there is an extension of $h$
to any of \( d \)'s disjuncts). At every step, the machine asks the oracle if further chase steps apply and goes on to guessing the next step if the answer is “yes”. The oracle is guaranteed to answer “no” after polynomially many invocations (in the size of \( Q_i \)), due to the first item in theorem D.1.1.

Once the leaf \( L_j \) is guessed, the machine checks conditions (1) and (2) in PTIME (in the size of \( L_j \) which is polynomial in that of \( Q_i \), hence also in that of \( p_1 \)) and answers “yes” if any of them is true. Otherwise, it checks condition (3) by asking the oracle (this can be checked in NP in the maximum size of a DED, as finding containment mappings is in NP). The machine answers ”yes” if and only if the oracle answers ”no”.

(2) Note that in the absence of disjunction of any kind, \( p_1 \) is translated to a single conjunctive query, \( Q_1^1 \). The chase tree degenerates into a single root-leaf path, because there is no disjunction in \( \Sigma_C \) and because the absence of the element equality tests and ancestor and ancestor-or-self navigation steps guarantees that (line) in TIX never applies. This single root-leaf path corresponds to a standard chase sequence, whose result is a conjunctive query we denote with \( L_1 \). By the first item of theorem D.1.1, the number of steps in this chase sequence is polynomial in the size of \( Q_1^1 \). For each step in the sequence, the machine must guess a homomorphism from some dependency \( d \in \Sigma_C \cup \text{TIX} \), which is polynomial in the size of \( d \). Once the chase sequence has been guessed, the machine checks conditions (1) and (2) from theorem D.1.1 in PTIME, and if none is satisfied, it guesses a containment mapping from \( Q_2^1 \) into the chase result (polynomial in the size of \( Q_1^1 \)).

(3) As in (2), the absence of disjunction ensures that the paths are translated to the single conjunctive queries \( Q_1^1, Q_2^1 \), and, together with the absence of equality tests, this ensures that the chase of \( Q_1^1 \) degenerates to a sequence. The chase result \( L_1 \) is polynomial in the size of \( Q_1^1 \).

Let \( I(L_1) \) be a TIX-instance obtained from \( L_1 \) such that (i) \( \mathbf{el} \) consists of all variables and constants in \( L_1 \), (ii) the entries in \text{child, attr, tag, text, id} are the corresponding atoms in \( L_1 \)'s body, and (iii) \( \mathbf{desc} \) is the minimal relation closed under (base), (trans), (refl). It is easy to see that \( I(L_1) \) can be computed in PTIME in the size of \( L_1 \). It is also easy to show that there is a containment mapping from \( Q_1^2 \) into \( L_1 \) if and only if \( Q_1^1 \)'s head variable belongs to the result of evaluating \( Q_2^1 \) on \( I(L_1) \). But the latter evaluation can be performed in PTIME in both the size of \( I(L_1) \) (hence \( L_1 \)) and of \( Q_2^1 \). This is because the absence of
equality tests makes $Q_2$ an *acyclic* query, for which Yannakakis shows PTIME evaluation
(in the combined expression and data complexity) [AHV95].

We therefore only need to guess the homomorphisms for the chase steps, which can be
done in NP in the size of the dependencies. But in the absence of (or if we fix) the SXICs
in $C$, we can find any homomorphism from a dependency $d$ in PTIME in the size of $Q_1^1$
by simply trying all mappings (their number is exponential only in the size of $d$).

**Remarks.** In practice the decision procedure from theorem D.1.1 is typically invoked
repeatedly to check containment under the *same* set $C$ of bounded SXICs. In this scenario,
we can consider $C$ fixed, in which case the complexity bounds in the theorem are only in
the size of the simple XPath expressions. In particular, if $C = \emptyset$ (there are no integrity
constraints), we obtain upper bounds for containment over *all* XML documents.

Note that if we disallow disjunction, containment is in NP, and thus no harder than
for relational conjunctive queries. We will see in section D.2 that this situation changes
for extensions of simple XPath expressions: adding navigation to wildcard children or to
ancestors raises complexity of containment to $\Pi_2^p$-hard (theorem 9.3.1) even in the absence
of disjunction!

**D.1.2 Undecidability**

In practice, we often know that XML documents satisfy SXICs that are not necessarily
bounded, the most salient examples being SXICs implied by DTDs, such as $(\text{someAddress})$
from section 9.1. Unfortunately, we have the result in theorem 9.2.2 showing undecidability
of containment.

**Proof of theorem 9.2.2:** By reduction from the following undecidable problem: Given
context-free grammar $G = (\Sigma, N, S, P)$ where $\Sigma$ is the set of terminals (containing at least
two symbols), $N$ the nonterminals, $S \in N$ the start symbol, $P \subseteq N \times (\Sigma \cup N)^*$ the
productions, and $L(G)$ the language generated by $G$, the question whether $L(G) = \Sigma^*$ is
undecidable [HU79].

**Note.** For the sake of presentation simplicity, the reduction we show below is to
containment in the presence of bounded SXICs and DTDs. However, a careful analysis of
the used DTD features reveals that these are captured as SXICs of two forms: $\forall x [/\!// A x \rightarrow$
The reduction. Given context-free grammar \( G = (\Sigma, N, S, P) \), we construct an instance \((\text{DTD}_G, D_G, XP_1 \subseteq XP_2)\) such that \(XP_1\) is contained in \(XP_2\) over all XML documents conforming to the description \(\text{DTD}_G\) and satisfying the dependencies in \(D_G\) if and only if \(\Sigma^* \subseteq L(G)\). We first show \(\text{DTD}_G\), which does not exercise all features of DTDs. The features of \(\text{DTD}_G\) used to prove undecidability can be easily shown to be fully captured by SXICs:

\[
\text{<!ELEMENT B (A|E)> } \quad \text{<!ATTLIST B } \quad \text{<!ATTLIST A}
\]

\[
\text{<!ELEMENT A (A|E)> } i \ #ID, \quad i \ #ID,
\]

\[
\text{<!ELEMENT E (PCDATA)> } S \ #IDREFS> \quad \text{sym (a1|a2|...|an)},
\]

\[
\text{N1 \ #IDREFS}, \quad \ldots \quad \text{Nk \ #IDREFS>}
\]

\(B, E, A\) are fresh names, \(a_1, \ldots, a_n\) are the alphabet symbols in \(\Sigma\), \(N_1, \ldots, N_k\) are the nonterminals in \(N\). Every document conforming to \(\text{DTD}_G\) is a list (unary tree) of elements, whose head is tagged \(B\) and unique leaf tagged \(E\). The inner elements (if any) of the list are tagged \(A\), and their \(\text{sym}\) attribute contains a symbol of \(\Sigma\). Every document conforming to \(\text{DTD}_G\) thus corresponds to a word \(w \in \Sigma^*\), and every pair \(s, t\) of \(A\)-elements such that \(t\) is a descendant of \(s\) determines a substring of \(w\).

The set of dependencies \(D_G\) (shown shortly) is designed such that, whenever a document conforms to \(\text{DTD}_G\) and satisfies \(D_G\), the following claim holds: for every pair \(s, t\) of \(A\)-elements with \(t\) a descendant of \(s\), let \(u\) be the corresponding substring of \(w\) (if \(s = t\), \(u\) is the unit length string given by the value of \(t\)'s \(\text{sym}\) attribute). Then for every \(1 \leq j \leq k\) such that there is a derivation of \(u\) starting from nonterminal \(N_j\), the value of the attribute \(i\) of \(t\) is a token of the value of the \(N_j\) attribute of \(s\). Furthermore, the \(S\) attribute of the \(B\)-element contains all tokens of the \(S\) attribute of the first \(A\)-element, if any.

We omit the proof of the claim, but illustrate for the grammar \(S \rightarrow cS | cc\) and word

\[1\text{Recall that an IDREFS attribute}\ a\ \text{models a set of IDREF attributes, represented by the set of whitespace-delimited tokens of}\ a\text{'s string value.}\]
\( w = \text{ccc} \). An XML document corresponding to \( w \) which conforms to DTD_G and satisfies the claim is

\[
\langle \text{B} \ i='0' \ S='2 \ 3' \rangle
\langle \text{A} \ \text{sym}='c' \ i='1' \ S='2 \ 3' \rangle
\langle \text{A} \ \text{sym}='c' \ i='2' \ S='3' \rangle
\langle \text{A} \ \text{sym}='c' \ i='3' \ S='.' \rangle
\langle \text{E} \rangle \text{any text goes here} \langle /\text{E} \rangle \langle /\text{A} \rangle \langle /\text{A} \rangle \langle /\text{A} \rangle \langle /\text{B} \rangle
\]

Now we have \( w \in L(G) \) if and only if there is a derivation of \( w \) in \( G \) starting from \( S \), which by the claim is equivalent to the \( i \)-attribute in the parent of the \( \text{E} \)-element being among the tokens of the \( S \)-attribute in the \( \text{B} \)-element. Therefore, \( \Sigma^* \subseteq L(G) \) is equivalent to the containment

\[
//.//\text{E}[@i] \subseteq /\text{B}[@S]
\]

which we pick for \( XP_1 \subseteq XP_2 \).

We now show the dependencies \( D_G \). For every production \( p \in P \), we construct a dependency \((prod_p)\) as illustrated by the following example. Let \( R, T \) be nonterminals and \( a, b \) alphabet symbols in the production \( R \rightarrow aRbT \). The corresponding dependency is

\[
(prod_p) \ \forall x, y \ [ x .//S[@sym='a']/@id(@R)/S[@sym='b']/@id(@T)/@i \ y \rightarrow x .//@R \ y]
\]

We enforce that the tokens in the \( S \)-attribute of the first \( \text{A} \)-element be included in the \( S \)-attribute of the \( \text{B} \)-element with the SXIC

\[
(start_B) \ \forall x, y \ [ /\text{B} \ x \land x .//@S \ y \rightarrow x .//@S \ y]
\]

Furthermore, we may assume without loss of generality that \( G \) has at most one \( \epsilon \)-production, namely \( S \rightarrow \epsilon \) (see the procedure for elimination of \( \epsilon \)-productions employed when bringing a grammar in Chomsky Normal Form [HU79]). If \( S \rightarrow \epsilon \in P \), add to \( D_G \) the SXIC

\[
(d_\epsilon) \ \forall x, y \ [ /\text{B} \ x \land x .//@i \ y \rightarrow x .//@S \ y]
\]
Remark. The undecidability result of theorem 9.2.2 does not preclude us from using the procedures in theorem D.1.1 and section D.2 for checking containment even under arbitrary SXICs. If the chase terminates, then containment holds if and only if and any of the conditions (1),(2),(3) in theorem D.1.1 are satisfied. The problem is that for arbitrary SXICs the chase may diverge. We can always impose a threshold after which we stop the chase and check the conditions. This would result in a sound, but incomplete procedure for checking containment. Our experience with the chase for the relational/OO data model [PDST00] suggests that there are many practical cases in which the chase terminates even if the SXICs are not bounded.

D.2 Detailed Treatment for Extensions of Simple XPath

Note that the translation of enriched XPath expressions is compatible with that of simple XPath expressions, and the addition of the wildcard child, parent and ancestor navigation is a very natural extension, which doesn’t even require new schema elements in TIX. We chose to handle these extensions separately because, innocuous as they may seem, they change complexity bounds dramatically. It turns out that the dependencies in TIX become insufficient in reasoning about wildcard expressions. Here is a counterexample to theorem D.1.1.

Example D.2.1 There are simple XPath expressions $p, p'$ extended with wildcard child navigation such that $p$ is contained in $p'$ but $T(p)$ is not contained in $T(p')$ under TIX:

\[
p = /\{ b/1[@x = "1"] \text{ and } \\
\quad a[a[@x = "1" \text{ and } */a \text{ and } c] \text{ and } */*[c \text{ and } */* \text{ and } @x = "0"] ] \\
\quad b/0[@x = "0"] \}
\]
\[
p' = /\{ ./[a[a \text{ and } c] \text{ and } \\
\quad */*[c \text{ and } */* \text{ and } @x = /b/*[@x]] ]
\]

In case the reader finds the graphical representation useful, we refer to Figure D.2, in which we depict child navigation steps with single arrows and descendant navigation.
steps with double, dashed arrows. The tag names are used to label the nodes (* is used for wildcards), and solid non-arrow lines associate attributes with nodes. @x = 0 indicates that the string value of the x-attributes is “0”. The dotted line represents an equality condition on x-attributes.

To see that p is contained in p', observe that a/a in p is equivalent to a/a ∪ a/*//a, and hence p is equivalent to p1 ∪ p2 where p1, p2 are obtained by replacing the subpath a/a with a/a, respectively a/*//a in p. But both p1, p2 are contained in p', as witnessed by the containment mappings matching the x-attributes in p' against the “0”-valued x-attributes of p1, respectively the “1”-valued x-attributes of p2.

On the other hand, according to the chase theorem [AHV95], T(p) is not contained in T(p') under TIX because there is no containment mapping from T(p') into chase_{TIX}(T(p)). Intuitively, what TIX does not capture is the minimality of desc: it only states that the latter contains the reflexive transitive closure of child, but it doesn’t rule out pairs of nodes that aren’t reachable via child navigation steps. TIX-instances containing such a pair (s, t) ∈ N × N are counterexamples for the containment: subpath */a in p is satisfied by the nodes r, q where child(r, s), desc(s, q), tag(q, a) even if s has no immediate child, while ./ */a in p' is not.

It turns out however that theorem D.1.1 holds if p1, the contained wildcard XPath expression, is //free.

We will use this observation to extend our decision procedure to handle wildcard expressions. First, we introduce some notation. Observe that any //free XPath expression is equivalent to a finite union of ancestor-or-self-free and ancestor-free expressions.
For instance, /a/b/ancestor-or-self is equivalent to (/a/b ∪ /a/b/.. ∪ /a/b/../..). There is no point in instantiating the occurrence of ancestor-or-self to more parent navigation steps (..) since the resulting expression would be unsatisfiable, that is empty over all documents. We denote the set of ancestor-free and ancestor-or-self-free paths in this finite union with \( \text{af}(p) \).

**Proposition D.2.2** Let \( C \) be a set of tree SXICs, let \( p \) be a //free wildcard XPath expression, and let \( \text{af}(p) = \{p_1, \ldots, p_n\} \). Then \( p \) is contained in wildcard expression \( p' \) under \( C \) if and only if both items of theorem D.1.1 are satisfied when substituting \( p_i \) for \( p_1 \) and \( p' \) for \( p_2 \), for every \( 1 \leq i \leq n \).

Recall that tree SXICs are restricted bounded SXICs, so the chase with them is defined. By proposition D.2.2, the decision procedure for containment of simple XPath expressions given in theorem D.1.1 can be used to decide containment of //free wildcard XPath expressions under tree SXICs.

We next show how to use proposition D.2.2 to decide containment even if \( p \) contains navigation along the descendant axis. First, observe that // = \( \bigcup_{0 \leq k} *^k \), where \( *^k \) is short for the concatenation of \( k \) wildcard navigation steps. More generally, every wildcard XPath expression \( p \) with \( n \) occurrences of // is equivalent to an infinite union of //free queries: denoting with \( p(k_1, \ldots, k_n) \) the result of replacing the \( i^{th} \) occurrence of // in \( p \) with the concatenation of \( k_i \) wildcard navigation steps, \( p \) is equivalent to \( \bigcup_{0 \leq k_1, \ldots, 0 \leq k_n} p(k_1, \ldots, k_n) \).

Therefore, the containment of \( p \) in \( p' \) reduces to checking the containment of each \( p(k_1, \ldots, k_n) \) in \( p_2 \), which is done according to proposition D.2.2. This still doesn’t give us a decision procedure, since there are infinitely many containments to be checked. The key observation to our containment decision procedure is that it is sufficient to check the containment for only finitely many //free queries in the union. For arbitrary \( p \), we denote with \( \text{wts}(p) \) the wildcard tag size, i.e. the number of \( * \) navigation steps in \( p \). For instance, \( \text{wts}(/a/ * /b/..c/@*) = 1 \) (note that wildcard attributes @* are not counted). Furthermore, we denote with \( \text{ps}(p) \) the parent size, i.e. the number of .. navigation steps in \( p \). Recalling that ancestor is syntactic sugar for ../ancestor-or-self, this means we count ancestor navigation steps as well: \( \text{ps}(/a/../ancestor) = 2 \).
Proposition D.2.3  Let $C$ be a set of bounded SXICs, $p_1, p_2$ be wildcard XPath expressions and let $l \overset{\text{def}}{=} \text{wts}(p_2) + \text{ps}(p_2) + \text{ps}(p_1) + 1$. Then
\[
p_1 \subseteq_C p_2 \iff \bigcup_{0 \leq k_1 \leq l, \ldots, 0 \leq k_n \leq l} p_1(k_1, \ldots, k_n) \subseteq_C p_2
\]

This result gives us the following decision procedure for containment of $p_1$ in $p_2$:

**Step 1:** We first translate away the disjunction ($|$ and $\text{or}$), obtaining finite unions $U_1, U_2$ of XPaths.

**Step 2:** We next use proposition D.2.3 to obtain from $U_1$ a finite union of $//\text{-free}$ queries $DF_1$, which must be checked for containment in $U_2$.

**Step 3:** Containment of $DF_1$ in $U_2$ is decided using the following easy result:

Proposition D.2.4  The union of $//\text{-free}$ wildcard XPath expressions $\bigcup_{i=1}^{m} p_i$ is contained in the union of wildcard XPath expressions $\bigcup_{j=1}^{m} p'_j$ under the bounded SXICs $C$ if and only if for every $1 \leq i \leq n$ there is a $1 \leq j \leq m$ such that $p_i \subseteq_C p'_j$.

**Step 4:** Finally, checking each $//\text{-free}$ $p_i$ for containment in $p'_j$ is done using proposition D.2.2.

Given this decision procedure, the proofs of theorems 9.3.2 and 9.3.3, claiming $\Pi_2^p$ upper bounds for diverse extensions of simple XPath are straightforward adaptations of the proof of theorem D.1.1. We illustrate for the case of simple XPaths with wildcard child navigation under tree SXICs:

**Proof of theorem 9.3.3:** We prove equivalently that non-containment is in $\Sigma_2^p$, that is it is decidable by an NP machine with an NP oracle. By proposition D.2.3, it is enough if the machine exhibits a $//\text{-free}$ query in the finite union which is not contained in $U_2$. To this end, the machine guesses $p_1 \in DF_1$ and $p_2 \in U_2$, computes $l$, guesses $0 \leq k_1, \ldots, k_n \leq l$, guesses $p$ in $af(p_1(k_1, \ldots, k_n))$ and next continues like in the proof of theorem D.1.1. •

Given the presence of $|$ and $\text{or}$ in the fragment of wildcard XPath expressions, it is not surprising that the algorithm is asymptotically optimal (we’ll see shortly that its lower bound is $\Pi_2^p$ as well): [SY80] shows that containment of conjunctive queries with
union is $\Pi_2^P$-complete. The upper bound however does not follow from [SY80]: the decision procedure of [SY80] works in the absence of dependencies, and hence must be extended to work under the ones in TIX.

However, we prove a stronger result in theorem 9.3.1, showing $\Pi_2^P$-hardness even for containment of disjunction-free extensions of simple XPath.

**Proof of theorem 9.3.1:** We only show the proof for the extension with wildcard child navigation, which is the most interesting one.

The proof is by reduction from the $\Pi_2^P$-complete $\forall \exists 3 - SAT$ problem [Pap94]: the instances of this problem are first-order sentences $\phi$ of general form $\forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m \land_{i=1}^l C_i$, where each clause $C_i$ is a disjunction of three literals which are any of the variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ or their complements. $\phi$ is a "yes" instance if and only if it is valid.

For every instance $\phi$, we construct the instance $p_1 \subseteq p_2$, where $\phi$’s variables appear as attribute and variable names, and $p_1, p_2$ contains occurrences of $@x_i$, $\$x_i$ for every $1 \leq i \leq n$, and occurrences of $@y_j$, $\$y_j$ for every $1 \leq j \leq m$. We use the notation $p_1(k_1, \ldots, k_n)$ introduced for proposition D.2.3. The containment holds if and only if $p_1(k_1, \ldots, k_n) \subseteq p_2$ for all $0 \leq k_i$. We claim that the reduction is defined such that the latter holds if and only if $\phi$ has a satisfying assignment which makes $x_i$ false if $k_i = 0$, and true if $k_i > 0$. This makes $\phi$ valid if and only if $p_1 \subseteq p_2$. The claim is proved after we give the construction.

Both $p_1, p_2$ return either the root of the document itself, or an empty node set: they have the form $/[q_1]$ where $q_1, q_2$ are qualifiers. $q_1$ is constructed as the conjunction of $7l + m + n$ subexpressions:

- for every clause $C_i$, let $u_i, v_i, w_i$ be its variables, and $a_{i,1}, \ldots, a_{i,7}$ the seven satisfying assignments for $C_i$. For every $1 \leq i \leq l$ and $1 \leq j \leq 7$, $q_1$ contains the subexpression $C_i[@u_i = a_{i,j}(u_i), @v_i = a_{i,j}(v_i), @w_i = a_{i,j}(w_i)]$.
- for every $y_j$, we add the existential gadget $y_j[@y_j = "0" \text{ and } @y_j = "1"]$ to $q_1$.
- We also add $n$ copies of a universal gadget (one copy for every $x_i$). The universal gadget (defined shortly) is denoted $U(l)$ and it is a wildcard XPath subexpression
having occurrences of \(@l\) for some attribute name \(l\). For every \(x_i\), the corresponding copy of \(U\) has \(@l\) substituted with \(@x_i\), denoted \(U(x_i)\).

This completes the construction of \(q_1\), up to the specification of the universal gadget. First we show the construction of \(q_2\), which contains \(l + m + n\) subexpressions:

- for every \(1 \leq i \leq l\), \(q_2\) contains \(C_i[\@u_i = \$u_i, \@v_i = \$v_i, \@w_i = \$w_i]\) where, as before, \(u_i, v_i, w_i\) are the variables occurring in clause \(C_i\). Note how they give both the names of the attributes and the names of variables.

- for every \(1 \leq j \leq m\), \(q_2\) contains the subexpression \(y_j[\@y_j = \$y_j]\).

- for every \(1 \leq i \leq n\), \(q_2\) contains a copy of a satisfaction gadget (defined shortly). The satisfaction gadget is denoted \(S(l)\) and it is a wildcard XPath subexpression with occurrences of \(@l\) and \($l\) for some \(l\). For every \(x_i\), \(q_2\) contains a copy \(S(x_i)\) in which \(@l, $l\) are substituted by \(@x_i, $x_i\).

We exemplify the construction so far on a \(\forall \exists 2 - SAT\) instance for simplicity sake:

\[
\phi = \forall x \forall y \exists z (\bar{v} \lor \bar{w}) \land (x \lor y) \\
P = /\{ C_1[\@x = "0" \land \@y = "0"] \land C_1[\@x = "1" \land \@y = "0"] \\
    \land C_1[\@x = "1" \land \@y = "1"] \land \\
    C_2[\@y = "0" \land \@z = "1"] \land C_2[\@y = "1" \land \@z = "0"] \\
    \land C_2[\@y = "1" \land \@z = "1"] \land \\
    z[\@z = "0" \land \@z = "1"] \land \\
    U(x) \land U(y) \} \\
P' = /\{ C_1[\@x = \$x \land \@y = \$y] \land C_2[\@y = \$y \land \@z = \$z] \\
    \land z[\@z = \$z] \land S(x) \land S(y) \} \\

We now specify the universal and satisfaction gadgets. Recalling counterexample D.2.1, \(U(l)\) is a copy of of \(p\), with \(x\) acting as \(l\), and \(S(l)\) is a copy of \(p'\), with \(x\) acting as \(l\).

We still have to prove the claim. According to proposition D.2.3, \(p \subseteq p'\) if and only if \(p(k_1, \ldots, k_n) \subseteq p'\) both for \(k_i = 0\) and \(k_i > 0\). Recalling the discussion in counterexample D.2.1, the containment mapping corresponding to \(k_i = 0\) binds \($x_i\) to “0”, while that
corresponding to \( k_i > 0 \) binds \( \$x_i \) to “1”. Moreover, it is easy to see that any containment mapping from \( p' \) to \( p \) corresponds to a satisfying assignment of \( \phi \). Therefore, \( p_1 \subset p_2 \) if and only if every truth assignment to the \( x_i \)'s has an extension to the \( y_j \)'s that satisfies all clauses of \( \phi \) (or, equivalently, if and only if \( \phi \) is valid). •

**Remark.** It is interesting to see that the \( \Pi^0_2 \) lower bound is reached even in the absence of disjunction when both \( // \)- and \( * \) navigation steps are allowed. We found this result to be counterintuitive, as the presence of \( // \) or \( * \) in isolation results in NP-complete containment complexity (item (2) in theorem 9.2.1). It is only their *interaction* that increases complexity. The intuition behind this is the fact that \( // \) expresses disjunction implicitly in the presence of \( * \): \( //a \) is equivalent to \( (/a) \mid (/ * /a) \).