ENABLING DATA INTEROPERABILITY WITH PROVENANCE AND SCHEMA INTEGRATION

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Laura Chiticariu

September 2008

The Dissertation of Laura Chiticariu is approved:

__________________________
Professor Wang-Chiew Tan, Chair

__________________________
Professor Dimitris Achlioptas

__________________________
Professor Phokion G. Kolaitis

__________________________
Professor Renée J. Miller

__________________________
Professor Neoklis Polyzotis

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies
Table of Contents

List of Figures vi
List of Tables viii
Abstract ix
Dedication xiii
Acknowledgments xiv

1 Introduction 1
1.1 Common Mapping Formalisms . . . . . . . . . . . . . . . . . . . . . . . . . 7
1.2 Systems for Tracing Data Provenance over Mappings . . . . . . . . . . . . 9
  1.2.1 DBNotes: Tracing Data Provenance over SQL Queries . . . . . . . . . 13
  1.2.2 SPIDER: Tracing Data Provenance over Schema Mappings . . . . . . 17
1.3 A Schema Integration System . . . . . . . . . . . . . . . . . . . . . . . . . 21
1.4 Organization of the Dissertation . . . . . . . . . . . . . . . . . . . . . . . . 24

2 The DBNotes System 25
2.1 pSQL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  2.1.1 The Custom Propagation Scheme . . . . . . . . . . . . . . . . . . . . 35
  2.1.2 The Default Propagation Scheme . . . . . . . . . . . . . . . . . . . . 36
  2.1.3 The Default-All Propagation Scheme . . . . . . . . . . . . . . . . . . 38
2.2 Generating a Query Basis . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
  2.2.1 Correctness of Generate-Query-Basis . . . . . . . . . . . . . . . . . . 44
2.3 Generating a Query Basis for General pSQL Queries . . . . . . . . . . . . . 55
  2.3.1 Correctness of Generate-Containment-Basis . . . . . . . . . . . . . . 58
2.4 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
  2.4.1 A Naive Storage Scheme . . . . . . . . . . . . . . . . . . . . . . . . . 63
  2.4.2 The Translator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
  2.4.3 The Postprocessor . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
  2.4.4 Other Features of the Prototype . . . . . . . . . . . . . . . . . . . . . 68

iii
## Table of Contents

2.5 Experimental Evaluation .................................................. 71
   2.5.1 Methodology ......................................................... 73
   2.5.2 Experimental Results ............................................. 74
2.6 Discussion ........................................................................... 90
   2.6.1 Other Possible Schemes for Managing Annotations .......... 90
   2.6.2 Extensions ............................................................... 91
2.7 Related Work ...................................................................... 93

3 The SPIDER System ................................................................. 97
   3.1 Background ................................................................. 100
      3.1.1 Example Debugging Scenarios ................................. 104
   3.2 Route Algorithms ......................................................... 108
      3.2.1 Computing All Routes ............................................ 111
      3.2.2 Computing One Route ............................................ 125
   3.3 Extending to the Nested Relational Model ......................... 138
      3.3.1 The Nested Relational Model ................................. 138
      3.3.2 Computing Routes in the Nested Relational Model .... 143
   3.4 System Architecture ....................................................... 145
      3.4.1 Other Features of the Prototype ............................... 146
   3.5 Experimental Evaluation .................................................. 151
      3.5.1 The Synthetic Datasets ............................................ 153
      3.5.2 The Real Datasets .................................................. 159
   3.6 Related Work .................................................................. 161
      3.6.1 Approaches in Deductive Databases ......................... 162
      3.6.2 Computing the Provenance of Data ......................... 164

4 The Schema Integration System .............................................. 169
   4.1 Integration through Concepts ........................................... 174
      4.1.1 Concepts .............................................................. 177
      4.1.2 Extracting Concepts from Schemas ......................... 180
      4.1.3 Matching the Concepts .......................................... 181
   4.2 Merging the Concepts ...................................................... 184
      4.2.1 The ApplyAssignment Algorithm ............................ 185
      4.2.2 Mapping Generation ............................................. 195
      4.2.3 From Concepts to Integrated Schema ....................... 200
   4.3 Preservation Properties .................................................... 202
   4.4 Enumeration of Alternatives ............................................ 204
      4.4.1 Duplicates and Cycles .......................................... 205
      4.4.2 Duplicate-Free Enumeration Algorithm .................... 207
      4.4.3 Adaptive Enumeration .......................................... 210
   4.5 System Architecture ....................................................... 213
   4.6 Experimental Evaluation .................................................. 217
      4.6.1 The Synthetic Scenarios ........................................ 218
## List of Figures

1.1 Recurrent scenarios in applications requiring data interoperability: (a) data exchange, and (b) data integration. ................................................................. 2
1.2 A data interoperability scenario from the scientific domain. ................. 5
1.3 Approaches in tracing data provenance: (a) the eager approach; (b) the lazy approach. (c) A classification of systems for tracing data provenance where our contributions are highlighted. ......................................................... 10
1.4 Overview of our schema integration methodology. ............................... 22
2.1 Three scientific databases, a mapping table and the result of three pSQL queries. 31
2.2 Some of the auxiliary queries generated by Step 2 of Generate-Query-Basis on Example 2.2.1. .................................................................................. 42
2.3 Architecture of DBNotes ........................................................................ 63
2.4 Snapshot from the visual interface of DBNotes. ..................................... 68
2.5 Visual diagrams displayed by DBNotes for (a) provenance, and (b) flow of data. 70
2.6 (a) Queries used in our experiments and (b) comparison in performance for default and default-all schemes on the 100MB dataset. ....................... 75
2.7 Comparison in performance for default and default-all schemes on the (a) 500MB, and respectively (b) 1GB dataset. ......................................................... 76
2.8 Performance comparison for (a) default and (b) default-all pSQL queries on databases annotated in various degrees on the 100MB dataset. .......... 79
2.9 Performance comparison for (a) default and (b) default-all pSQL queries on databases annotated in various degrees on the 500MB dataset. .......... 80
2.10 Performance comparison for (a) default and (b) default-all pSQL queries on databases annotated in various degrees on the 1GB dataset. .............. 81
2.11 Performance comparison for default-all queries on the 100MB dataset when the number of join attributes selected in the output is varied. .......... 83
2.12 Comparison in performance for default and default-all schemes on the 100MB database annotated 200%. ......................................................... 87
2.13 Performance comparison for default (a) and default-all (b) pSQL queries on 100MB databases with more than 100% annotations. ...................... 88
2.14 Classification of systems for tracing the provenance of data over SQL queries. 93
## List of Tables

2.1 The execution time of each query for each database in the 100MB dataset and each propagation scheme. The columns “#pSQL” and “#SPJ” denote the size of the query basis and number of SPJ queries that are generated, respectively, for the default-all scheme. ........................................... 77

2.2 The average percentage increases incurred in the total execution times of the queries under both default and default-all schemes when the number of annotations in the database is varied from 30% to 60% annotations and from 60% to 100% annotations. ................................................................. 82

2.3 The execution time of each default-all query from Experiment 3 on the 30%, 60% and 100% annotated databases in the 100MB dataset. The columns “#pSQL” and “#SPJ” denote the size of the query basis and respectively, the number of SPJ queries that are generated. The columns “#tuples” show the number of tuples retrieved by the queries before the postprocessing phase. ................. 84

2.4 The execution time of each query for each propagation scheme and each database annotated more than 100% in the 100MB dataset. The columns “#pSQL” and “#SPJ” denote the size of the query basis and number of SPJ queries that are generated, respectively, for the default-all scheme. ......................... 86

3.1 Real datasets and schema mappings characteristics. ........................................ 159

4.1 Some of the enumeration constraints used in experiments with real schema integration scenarios. ................................................................. 222
Abstract

Enabling Data Interoperability with Provenance and Schema Integration

by

Laura Chiticariu

Interoperability is the ability of a system to work with other systems without special effort on the part of the user. Achieving interoperability of independent data sources has been recognized as one of the “oldest database problems” [21]. However, despite the amount of research in this area, data interoperability continues to remain a challenge in practice. One cause of complexity in providing support for data interoperability is the high heterogeneity of the data. Interoperability of heterogeneous databases requires creating, managing and reasoning about complex relationships, called mappings, between these databases. This is a challenging, human-intensive and error-prone process in practice. The goal of this dissertation is to enable the interoperability of such heterogeneous data sources.

Among the contributions of this dissertation are two principled methods (and corresponding systems) for tracing the provenance of data that has been transformed through chains of mappings. Provenance of data describes the origins, as well as the journey of data, through multiple mappings, from its origins to its destination. This aspect of the dissertation enables data interoperability in that it facilitates a user’s understanding of the quality of the data, as well as the quality of the mappings, used in applications involving data from multiple sources.

We study the problem of tracing the provenance of data in the context of two commonly used mapping formalisms: SQL queries, and respectively, schema mappings. In the
DBNotes system [22, 23, 33], we develop and implement an approach for tracing data provenance over SQL queries. In the SPIDER system [8, 32], we develop and implement an approach for tracing data provenance over schema mappings.

DBNotes is an annotation management system for relational databases where each attribute value of each tuple in a relation is associated with zero or more annotations, and annotations propagate along with the data as it is being transformed through an SQL query. In the *default* propagation scheme, annotations propagate based on provenance. Hence, the provenance of data in the result of a query can be determined by analyzing its annotations. The default annotation propagation behavior is sensitive to query rewriting. Hence, DBNotes provides an alternative propagation scheme, called *default-all*, which propagates annotations according to all equivalent rewritings of a given query. A main result of this dissertation is that DBNotes can always simulate the default-all propagation behavior by executing finitely many queries, although there may be infinitely many equivalent rewritings. DBNotes also provides a third propagation scheme, called *custom*, which allows a user to specify how annotations should propagate. This part of the dissertation is based on joint work with Deepavali Bhagwat, Wang-Chiew Tan and Gaurav Vijayvargiya [22, 23]. A demonstration of the DBNotes system is joint work with Wang-Chiew Tan and Gaurav Vijayvargiya [33].

In contrast to DBNotes, which traces data provenance over mappings expressed as SQL queries, the SPIDER system traces data provenance over mappings expressed using a different formalism: the formalism of schema mappings. Schema mappings are high-level, declarative, assertions for specifying relationships between schemas. When compared to SQL queries, schema mappings have an advantage in that they operate at a higher level of abstraction.
Thus, they are easier to analyze and reason about. A fundamental difference between DBNotes and SPIDER is the method by which they compute provenance. DBNotes adopts the *eager* approach for computing provenance, which involves forwarding annotations along with the data to the output database. A problem with this approach is that it requires modifying the execution of the query to accommodate the annotations. In contrast, SPIDER adopts the *lazy* approach for computing provenance, which does not require any changes to the schema mapping execution engine. In tracing provenance over schema mappings, the eager approach has been studied before, whereas SPIDER is the only system which adopts the lazy approach in this context.

A primary feature of SPIDER is the novel notion of *routes*. Routes are a form of provenance: they describe the relationships between data in the source and the output databases that are established by the schema mapping. A main difficulty that arises in designing algorithms for computing routes is that schema mappings may be recursive. In this case, the number of routes may be exponential. Hence, it was initially unclear whether there exists a polynomial-time algorithm for computing routes. This difficulty has been resolved in SPIDER, which features two complete, polynomial-time algorithms for computing one or all routes for selected data, through chains of schema mappings that may be recursive. A main result of this dissertation states that SPIDER produces a compact, polynomial-size representation of all routes in polynomial time, even though there may be exponentially many routes. This part of the dissertation is based on joint work with Wang-Chiew Tan [32]. A demonstration of the SPIDER system is joint work with Bogdan Alexe and Wang-Chiew Tan [8].

Another contribution of this dissertation is a principled method and system for performing *schema integration*. Schema integration is the problem of designing a unified integrated
schema based on a set of overlapping source schemas. This aspect of the dissertation enables data interoperability in two ways. First, the integrated schema provides a standard representation of the source data, thus offering a way to deal with the heterogeneity in these databases. Furthermore, our system facilitates the process of designing mappings to relate the sources and the integrated schema, in that it automatically generates these mappings and guarantees they have precise information-preservation properties. The system features a principled method for enumerating multiple integrated schemas, and easy-to-use capabilities for refining the enumerated schemas via user interaction. Our method is a departure from previous approaches to schema integration, which do not offer a systematic exploration of the possible integrated schemas. This part of the dissertation is based on joint work with Phokion G. Kolaitis and Lucian Popa [31]. A demonstration of the schema integration system is joint work with Mauricio Hernández, Phokion G. Kolaitis and Lucian Popa [30].
To Alex and my parents
Acknowledgments

I am deeply grateful to my advisor, Wang-Chiew Tan. This work would not have been possible without her invaluable insights, wise advice and never ending patience. She has taught me the value of curiosity and rigorousness in doing research, and the importance of staying balanced, perhaps the most valuable life lesson of all. I am indebted to Wang-Chiew for allowing me to learn from her, and for having understood and kindly encouraged me, when balance seemed impalpable.

I am also grateful to my committee members, Dimitris Achlioptas, Phokion G. Kolaitis, Renée J. Miller and Neoklis Polyzotis, whose detailed comments and suggestions greatly helped in achieving the current form of this dissertation.

Over the past years, I had the opportunity to work with many wonderful people. I thank Howard Ho and Lucian Popa, my mentors in IBM, for offering me the opportunity to work with Clio, and sparking my interest in the schema integration problem. Their guidance and encouragement extended well beyond the summers I have spent as an intern in the Clio group at IBM Almaden, and fueled part of my research. I am also grateful to Phokion G. Kolaitis and Lucian Popa for their insightful comments and direction. The schema integration part of this dissertation is the result of our fruitful collaboration. I also thank Mauricio Hernández, and my colleagues Bogdan Alexe, Daniel Pepper, and Gaurav Vijayvargiya, with whom I have always had great fun in implementing various systems.

The Computer Science department at UC Santa Cruz has been a wonderful home for me in the past few years. In my research, I have benefitted from numerous comments and
suggestions received from the members of the Database Group, to whom I am very grateful. I
also thank eWomen, who provided me with great peer support over the years, and Tracie Tucker,
who made my life much easier by handling my administrative problems.

Last but not least, I would like to express my incommensurable gratitude to my family
- my parents and my sister - for their infinite love and encouragement in everything I had ever
decided to pursue, and to Alex Dragulescu, who has brought absolute happiness into my life.
Chapter 1

Introduction

Interoperability is the ability of a system to work with other systems, without special effort on the part of the user. In today’s information era, an important problem is providing support for data interoperability. A multitude of data sources reside within enterprises, the Web, or the scientific community. Often times, these data sources need to be transformed and combined in order to provide a novel, aggregated view of the data.

Within the enterprise domain, for example, interoperability between a company’s legacy database and its newer enterprise database is very desirable. Information from these two databases may be combined and used in applications such as a portal or an enterprise resource planning software. On the Web, online sellers such as Amazon combine data from thousands different vendors and present it to customers in a common format. Amazon must be able to quickly migrate information about available products from the vendors into its products database, and relay information about online purchases back to the vendors. Hence, interoperability between the databases of the vendors and Amazon’s products database is crucial. In
science, there is a multitude of databases that store results from scientific experiments (see [41] for a catalog of 500 biology databases). Interoperability of such databases is important in providing a scientist with a unified view of a specific problem.

\textit{Data exchange} [62] and \textit{data integration} [65] are two recurrent scenarios in applications requiring data interoperability. Figure 1.1(a) illustrates the data exchange scenario, where the goal is to transform data that conforms to one schema, called the source schema, into data that conforms to a different schema, called the target schema. Figure 1.1(b) illustrates the data integration scenario, where a unified virtual interface, called the global or integrated schema, is used to provide uniform access to a set of data sources. The problem of generating the integrated schema based on a set of overlapping source schemas is called \textit{schema integration}. In data integration, the source data is not materialized under the integrated schema. Instead, the goal is to retrieve and combine the desired data from the sources, by means of answering queries posed over the integrated schema.

The relationships between the schemas involved, called \textit{mappings}, are fundamental components in data exchange and data integration. In data exchange, the mapping between the
source and target schemas, denoted as $M$ in Figure 1.1(a), specifies how the source data is to be migrated to the target. In data integration, the mappings between the source schemas and the integrated schema, denoted as $M_1$ to $M_n$ in Figure 1.1(b), specify how the source data is to be reflected under the integrated schema. The integrated schema itself is another fundamental component in data integration. Together with the mappings $M_1$ to $M_n$, the integrated schema is crucial in ensuring the accuracy of the results retrieved from the sources, via queries posed over the integrated schema.

Achieving data interoperability has been recognized as one of the “oldest database problems” [21]. Both data exchange and data integration have been addressed as early as several decades ago in EXPRESS [85], a system for exchanging data between hierarchical databases, and, respectively, in MULTIBASE [86], a system for integrating distributed databases. Despite plenty of research in this area, however, achieving data interoperability continues to be a challenge in practice, as it has been recently pointed out by Haas [55].

“68% of the CEOs [surveyed in [1]] listed the integration of disparate applications and infrastructure as a key issue for their business, one that slows them down and stops the flow of information. Meanwhile, […] 30% of their peoples time is spent just looking for the information they need to do their jobs.”

Haas [55]

One cause of complexity in providing support for interoperability of data sources is the high heterogeneity of the data. Data sources reside in a multitude of formats, such as relational or XML, which leads to structural heterogeneity in these databases. Furthermore, data sources are usually independently created by different people who understand the domain
in various diverse ways, leading to semantic heterogeneity: even databases describing identical entities may carry completely different semantics. Interoperability of such heterogeneous data sources requires creating, managing and reasoning about complex mappings between these databases. This is a challenging, human-intensive and error-prone process in practice; it is a process in need of better solutions, as it has been recently pointed out by Bernstein and Melnik [21].

“Coping with data heterogeneity is one of the most time consuming data management problems. Anecdotal evidence suggests it is 40% or more of the work in enterprise IT departments. […] In short, it is a problem in need of better solutions.”

Bernstein and Melnik [21]

In what follows, we describe some of the challenges in achieving interoperability of heterogeneous databases with a simple scenario inspired from the scientific domain. Suppose GeneLab, a fictional bioinformatics laboratory, wishes to start a series of experiments to study the interaction between several genes and proteins for the development of a new medical compound. For this purpose, GeneLab wishes to gather sufficient information about the elements of interest before starting the experiments, and information from several different data sources may need to be combined in the process. For example, a few data sources of interest might be the protein databases UniProt [9] (formerly known as SWISS-PROT [12]) and PIR [13], and the GenBank [17] gene database. Another relevant data source is the database of XClinic, a fictive start-up which initiated a similar set of experiments before being acquired by GeneLab. The application designed in support of this scenario is illustrated in Figure 1.2. The design of this application involves the following steps. First, an integrated schema is designed to represent the desired information from the four data sources. The integrated schema is desirable because it
Figure 1.2: A data interoperability scenario from the scientific domain.

provides a uniform representation of the underlying data, which may be heterogeneous in both structure and semantics. Second, the mappings between the sources and the integrated schema are designed. The mappings, denoted $M_1$ to $M_4$ in Figure 1.2, specify the relationships between the source schemas and GeneLab’s integrated schema. Two possible scenarios may occur at this point. In the data exchange scenario, the source data is migrated and materialized under the integrated schema, according to the semantics of mappings $M_1$ to $M_4$. Alternatively, in the data integration scenario, data is not materialized under the integrated schema. Instead, data at the sources is accessed by posing queries against the integrated schema. These queries are automatically translated, via the mappings, into queries over the sources. For illustration purposes, we shall assume the first scenario where the source data is exchanged into GeneLab’s database.

The success of GeneLab’s experiments depends, among other factors, on the quality of the data used in the experiments. One factor which influences the quality of GeneLab’s data is the quality of the source databases themselves. While UniProt, PIR and GenBank consist of only original, reliable information, XClinic’s database has been created by combining original information with information from several other databases. A GeneLab scientist may naturally wish to understand the origins of the data she examines, as well as how the data has been
transformed, through mappings, before reaching GeneLab’s database. A second factor which influences the quality of GeneLab’s data is the quality of the application design itself. How well does the integrated schema reflect the source databases? Does it “miss” any information from the sources? Do $M_1$, $M_2$, $M_3$ and $M_4$ correctly combine and transform the source data under the integrated schema?

Among the contributions of this dissertation are two principled methods (and corresponding system implementations) for tracing the provenance of data that has been transformed through chains of mappings. Provenance of data essentially means the origins, as well as the journey of data, through multiple mappings, from its origins to its destination. This part of the dissertation enables data interoperability in that it facilitates the process of understanding the quality of the data, as well as the quality of the mappings used in data exchange and data integration systems. If an output piece of data (e.g., a tuple, or an attribute value in a tuple) is discovered to be erroneous, the origins of the error, whether in the source data, or in the mappings themselves, may be discovered by analyzing the provenance of that output data.

Another contribution of this dissertation is a principled method and system for performing schema integration. This part of the dissertation enables data interoperability in that it facilitates the process of designing the integrated schema in data integration systems. It also facilitates the process of designing mappings to relate the sources and the integrated schema: these mappings are automatically generated by the system and guaranteed to possess precise information-preservation properties [70]. Informally, this means that all information in the source databases is preserved via the generated mappings under the integrated schema. Naturally, our systems for tracing data provenance can be used in conjunction with the schema
integration system, to further enhance a user’s understanding of the generated mappings.

The rest of this chapter gives an overview of the methods and systems that we have developed, and summarizes our contributions. Before doing so, however, we shall explain the formalisms commonly used for expressing mappings in data exchange and data integration.

1.1 Common Mapping Formalisms

**SQL Queries** A scenario that occurs quite often in practice is that of exchanging and integrating information stored in relational databases. A common language for specifying mappings in this context is the standard SQL query language [2]. In fact, most commercial mapping systems such as Altova MapForce [67] and DataDirect Stylus Studio [89] use SQL queries for expressing mappings between relational databases. Various other query and transformation languages such as XQuery, XSLT, SQL/XML have been used to express mappings in the more general context where data is relational or hierarchical (XML).

**Schema Mappings** Another formalism for expressing mappings is that of schema mappings. Schema mappings differ from query languages, such as SQL or XQuery, in that they operate at a higher level of abstraction. Specifically, the schema mappings formalism is based on the language of tuple generating dependencies (tgds) (also known as global-and-local-as-view (GLAV) assertions), and equality generating dependencies (egds). Tgds and egds are classical concepts in database theory [3], where they have been used to express common types of database constraints such as keys, foreign keys or inclusion dependencies. More recently, tgds and egds have been widely used in the formal study of the semantics of data exchange [62] and
data integration [65] in the relational context. To illustrate tgds and egds, consider the binary relations $R(A, B)$ and $S(A, C)$. The logical formula $\forall x \forall y \forall z (R(x, y) \land R(x, z) \rightarrow y = z)$ is an egd which specifies a key constraint on the $A$ attribute of $R$: every two $R$ tuples which agree on the value of the $A$ attribute, must also agree on the value of the $B$ attribute. The formula $\forall x \forall y (R(x, y) \rightarrow \exists z S(x, z))$ is a tgd specifying a foreign key constraint involving the $A$ attributes of $R$ and $S$, respectively: for each tuple $t$ of $R$, there must exist a tuple $t'$ of $S$ such that $t$ and $t'$ agree on the value of their $A$ attributes, respectively. In the rest of this dissertation, the term schema mappings is reserved to refer to mappings expressed using the language of tgds and egds, whereas the term mappings is used to refer to transformation specifications in general (whether expressed as queries, Java code, or schema mappings).

Tgds and egds have been used in a systematic study of data exchange between relational databases [62]. In this context, a schema mapping is a triple $\mathcal{M} = (S, T, \Sigma_{st} \cup \Sigma_t)$, where $S$ is the source schema, $T$ is the target schema, $\Sigma_{st}$ is a set of source-to-target tgds (or s-t tgds, in short), and $\Sigma_t$ is the union of a set of target tgds with a set of target egds. Intuitively, the set of s-t tgds $\Sigma_{st}$ specifies that the materialized target instance contains facts implied by facts in the source instance, according to the dependencies in $\Sigma_{st}$. Second, the materialized target instance has to adhere to the constraints of the target schema, which are specified by $\Sigma_t$.

In data integration [65], s-t tgds have been widely used to express mappings between the source schemas and the integrated schema. In this context, s-t tgds correspond to sound global-and-local-as-view (GLAV) assertions in the terminology of [65]. GLAV is a generalization of local-as-view (LAV) and global-as-view (GAV) assertions used in data integration. The Select-Project-Join-Union fragment of SQL corresponds to exact GAV assertions [65].
An example of data exchange research prototype that is based on schema mappings is Clio [56]. An example of a recent commercial data exchange and integration system based on schema mappings is IBM Rational Data Architect (RDA) [81]. Both Clio and RDA operate with schema mappings based on an extension of tgds and egds to handle the hierarchical model [77, 100], in addition to the relational model. These systems automatically translate the schema mapping specification into an executable query such as SQL or XQuery. The executable query is essentially an implementation for the schema mapping: the output data is generated by running the executable over the source data. When compared to executable languages such as SQL or XQuery, schema mappings have the advantage that they are more high-level, and can be seen as an abstraction for more complex specifications (e.g., SQL or XQuery) which are used to execute the transformation. Thus, they are easier to analyze and reason about. In fact, various operators for manipulating schema mappings have been rigorously studied in the literature. Examples include the composition [20, 45, 66] and the inverse [10, 43, 46] operators.

1.2 Systems for Tracing Data Provenance over Mappings

As mentioned earlier, the quality of mappings is crucial in ensuring that the data exchange or data integration application has the intended behavior (i.e., data is accurately transformed or combined according to the intended semantics). Due to the heterogeneity of the databases involved, the process of designing mappings cannot be fully automated, and mapping generation is usually performed semi-automatically. First, an initial mapping is automatically generated by the mapping system used in designing the data exchange or integration applica-
Subsequently, a mapping designer is involved in correcting and refining the mapping, and disambiguating between alternative semantics for the mappings. In real-life scenarios, mappings can be large and therefore difficult to understand by humans. Therefore, a facility for explaining the behavior of mappings is very desirable.

The use of data in explaining mappings is only natural. In this context, data provenance naturally comes to mind, since it helps answer questions such as: How was a piece of data created in the materialized target instance? What are the origins of this piece of data in the source database? Here, the term “piece of data” may refer to a tuple, or an attribute value of a tuple in a relational database, or an element or fact in a hierarchical database. By providing
answers to these questions, provenance illustrates the behavior of the mapping with the data, and explains how the data is transformed and combined through mappings. Hence, provenance enables data interoperability in two ways. First, it facilitates the process of understanding the mappings used in data exchange and data integration applications. Second, it also facilitates the process of understanding the quality of data used in data exchange applications. If a piece of data in the materialized target instance is identified as incorrect, the provenance of that piece of data can reveal the origins of the error, whether in the mapping, or in the source data itself.

This part of the dissertation studies the problem of tracing the provenance of data which has been transformed through chains of mappings, for the case of mappings expressed using the two common formalisms described earlier: SQL queries, and respectively, schema mappings. We have studied two possible approaches for tracing provenance: the *eager approach* and the *lazy approach*. Both these approaches are useful in different contexts, as we shall explain shortly. We have developed a principled method for *eagerly* tracing data provenance over SQL queries [23], which is based on forwarding annotations along with the data through queries. Based on this method, we have built DBNotes [34], an annotation management system for relational databases. We have also developed a principled method for *lazily* tracing data provenance over schema mappings [32]. Based on this method, we have implemented SPIDER [8], a system for understanding and debugging schema mappings. In the rest of this section we shall describe the trade-offs between eager and lazy approaches for computing provenance, and summarize our contributions in the DBNotes and SPIDER systems.

**Approaches in Tracing Data Provenance: Eager versus Lazy** Figures 1.3(a) and 1.3(b) illustrate the two possible approaches for tracing provenance. In the *eager approach*, the mapping
is reengineered so that extra information is carried over to the output database during the transformation, to help answer provenance. Usually, the provenance of a piece of output data can be subsequently derived by examining the output database and the extra information. Thus, provenance is *eagerly* computed and immediately available in the output. The *lazy approach* differs from the eager approach in that provenance is computed only when needed, thus *lazily*, by examining the source data, the output data, and the mapping. In contrast to the eager approach, the lazy approach does not require the reengineering of the mapping for the purpose of carrying additional information to the output database.

Both eager and lazy approaches for tracing provenance have advantages, as well as disadvantages, and they are appropriate in different scenarios. Since additional information is carried over and stored along with actual data in the output database, an eager approach involves a performance overhead during the execution of the mapping, as well as a space overhead for storing the extra information in the output. However, it has the advantage that if the right additional information is propagated, provenance may be derived directly from the output database and the extra information, without examining the source database. Hence, an eager approach is useful in scenarios where the source data may become unavailable after the transformation.

The lazy approach does not require the reengineering of the mapping. Hence, it has the advantage that it can be readily deployed on an existing mapping system without changes to the system, and furthermore, it does not incur in any performance or storage overhead at mapping execution time. Thus, a lazy approach is useful when storage space is an issue, or it is not possible to modify the implementation of the mapping execution system. A disadvantage of the lazy approach is that deriving provenance usually involves sophisticated techniques for
reasoning about the source database, the output database, and the mapping. Hence, the lazy approach cannot be used if the source data becomes unavailable.

The idea of tracing provenance of data is not new and has appeared in various forms in the literature. Figure 1.3(c) shows a classification of existing systems for tracing data provenance. A detailed description of these systems, along with other relevant related work, is given in Sections 2.7 and 3.6. Prior to the development of our DBNotes system, both a lazy and an eager approach for tracing provenance over SQL queries had been proposed and implemented in [37, 38], and respectively, in Polygen [98]. DBNotes differs from Polygen in that our method for eagerly tracing provenance over SQL queries is more general, as we shall explain in Sections 1.2.1 and 2.7. Prior to the development of our SPIDER system, only an eager approach for tracing provenance over schema mappings had been proposed [96]. To date, however, the method that we have implemented in SPIDER is the only approach for lazily tracing provenance over schema mappings.

1.2.1 DBNotes: Tracing Data Provenance over SQL Queries

We have developed a principled method for eagerly tracing data provenance over SQL queries [23] that is based on forwarding annotations along with the data, as data is being transformed through queries. We have implemented this method in our DBNotes Database anNotation management system [34]. In DBNotes, each column of each tuple in each relation can be annotated with zero or more annotations, where an annotation can be any type of information about data, such as provenance, comments, or other types of metadata. Mappings are expressed using SQL queries, and annotations automatically propagate along, as data is being
transformed from one database to another through SQL queries. By default, DBNotes propagates annotations based on where data is copied from. As a consequence, if each column of each tuple in a database is annotated with its address, the provenance of data is *eagerly* propagated along with the data and immediately available in the output. Hence, one immediate application is to use these annotations to systematically trace the provenance of data. Even if the data had undergone several transformation steps, the origins of data through the transformation steps can easily be determined by examining the annotations.

We study three annotation propagation schemes that are motivated by different needs. The *default* scheme propagates annotations according to where data is copied from, i.e., based on provenance. Hence, if each piece of data is annotated with its address, and these annotations propagate along as the data is being transformed from the source to the output, the provenance of an output piece of data can be computed by examining its annotations. Here, a piece of data refers to a column (i.e., an attribute value) of a tuple in a database. The *default-all* scheme propagates annotations according to where data is copied from among all equivalent formulations of a given query. The *custom* scheme allows a user to specify how annotations should propagate. This scheme is useful, for example, when the user wishes to retrieve annotations from one source over another.

In DBNotes, we have implemented all three propagation schemes by extending a fragment of SQL. This extension is called pSQL. A pSQL query is essentially an SQL query extended with a *PROPAGATE* clause that would propagate annotations according to one of the schemes described above as data is being transformed. The implementation of DBNotes is based on an underlying annotation storage scheme where for each attribute of each relation,
there is an additional column that stores the annotations for that attribute. A translation algorithm translates a given pSQL query into one or more SQL queries against these underlying relations, and these SQL queries will retrieve the relevant annotations, according to the specified propagation scheme. In the context of the default-all propagation scheme, one of the main results in this dissertation shows how it is always possible to generate finitely many queries that can simulate the annotation propagation behavior of the set of all equivalent queries, which is possibly infinite.

We have conducted a series of experiments to evaluate the feasibility of propagating annotations with DBNotes. An encouraging result is that the performance of pSQL queries with default scheme is comparable to those of SQL queries (i.e., without annotation propagation), on databases where each attribute value has at most one annotation. Hence, using DBNotes for tracing provenance is feasible, since having one annotation per attribute value usually suffices for this purpose. However, the performance of a pSQL query with default-all scheme is worse compared to the performance of the same query under the default scheme. This is explained by the fact that multiple queries (and not just a single query) are usually executed to achieve the default-all behavior, and the degradation in performance is roughly proportional with the number of these executed queries. Such performance overhead may be acceptable in certain special cases (e.g., these queries execute very fast, or only a few queries are needed to simulate the default-all behavior). However, it would be interesting to investigate optimization techniques for achieving the feasibility of the default-all scheme in the general case.

Our experimental results also indicate that on databases where each attribute value has at least one annotation, the performance of pSQL queries starts to degrade. This suggests that
DBNotes’ scheme for storing annotations may not be the best suited on such databases. Trade-offs between different other annotation storage schemes have been investigated in [5, 50].

A classification of systems for eagerly and lazily tracing data provenance over SQL queries is shown in Figure 1.3(c). Prior to the development of DBNotes, the lazy approach had been proposed and implemented in [37, 38]. An eager approach had also been implemented in the Polygen system [98]. In Polygen, however, only information about which source relations a value is copied from is propagated along. In contrast, DBNotes is flexible in the amount of information that is carried along to the result (i.e., it could be the source relations, or the exact location within the source relations, or a comment on the data). Hence, although our main motivation was tracing data provenance, DBNotes has many other uses. For instance, annotations can be used to describe information about data that would otherwise have not been kept in a database. An error report or remarks about a piece of data may be attached and propagated along to other databases, thus notifying other users of the error or additional information.

Another difference between DBNotes and Polygen is that Polygen only supports the default propagation behavior, while DBNotes provides, in addition, the custom and the default-all propagation schemes. In fact, DBNotes is the first annotation management system that allows a user to specify how annotations should propagate (i.e., the custom scheme), and provides an annotation propagation scheme that is invariant under equivalent queries (i.e., the default-all scheme). Mondrian [49, 50] and Trio [4, 16, 83] extend DBNotes by allowing annotations to be placed on sets of attribute values in a tuple, and respectively, entire tuples.
1.2.2 SPIDER: Tracing Data Provenance over Schema Mappings

The previous section overviews our principled approach for *eagerly* tracing data provenance over SQL queries that we have implemented in the DBNotes system. DBNotes operates in special (and common) data interoperability scenarios where the databases involved are relational, and the mappings between these databases are expressed using SQL queries. This section summarizes our principled method for *lazily* tracing data provenance over schema mappings [32] that we have developed and implemented in SPIDER [8], a Schema mappings DEbugger system. In contrast to DBNotes, SPIDER works in more general data interoperability scenarios where databases can be relational or hierarchical, and the relationships between these databases are expressed as schema mappings, as opposed to SQL queries.

Recall from Section 1.1 that a schema mapping is a triple $\mathcal{M} = (S, T, \Sigma)$, where $S$ is called the *source schema*, $T$ is called the *target schema*, and $\Sigma$ is a set of assertions specifying the relationships between instances of the two schemas. In the data exchange scenario, $\Sigma$ is the union of a set $\Sigma_{st}$ of s-t tgds with a set $\Sigma_t$ of target tgds and target egds. In the data integration scenario, $\Sigma$ is essentially a set of s-t tgds relating the source schemas with the integrated schema, and there are no constraints on the integrated schema. Given an instance $I$ of the source schema, a *solution for $I$ under $\mathcal{M}$* is an instance $J$ of the target schema, such that $I$ together with $J$ satisfy the schema mapping $\mathcal{M}$. As mentioned earlier, schema mapping systems, such as Clio and RDA, take the specification of $\mathcal{M}$ and generate an executable query (e.g., SQL) that, when run on the source instance $I$, produces a solution $J$ for $I$ under $\mathcal{M}$.

In SPIDER, we have implemented a principled approach for tracing data provenance
over schema mappings that is based on the concept of routes. Routes are a form of provenance: they describe the relationships between source and target data with the schema mapping $\mathcal{M}$. More concretely, a route for a set of selected target tuples $J_s$ in $J$ describes one possible way in which the tuples in $J_s$ can be witnessed with some source data in $I$ and the schema mapping $\mathcal{M}$. Note that SPIDER’s notion of provenance described by routes applies to sets of tuples.

Routes have declarative semantics, based on the logical satisfaction of dependencies in $\mathcal{M}$. They are not tied to any procedural semantics associated with the transformation of data from the source to the output according to the schema mapping. Hence, the concept of routes applies to any data exchange system that is based on schema mappings. Furthermore, routes can also be applied in data integration systems. In this context, test source data may be exchanged under the integrated schema for the purpose of facilitating the user’s understanding of the mappings between the sources and the integrated schema.

In computing routes with SPIDER, the schema mapping $\mathcal{M}$ is fixed and the input consists of the source instance $I$, a solution $J$ for $I$ under $\mathcal{M}$, and a set of selected target data $J_s$ from $J$. In designing algorithms for computing routes, a main difficulty arises when the schema mapping $\mathcal{M}$ is recursive (i.e., the set of target dependencies of $\mathcal{M}$ is cyclic). In this case, there may be exponentially many routes. Hence, it was initially unclear whether there exists a polynomial time algorithm for computing routes. A main result in this dissertation is that SPIDER can compute a concise, polynomial-size representation of all routes, in time polynomial in the size of the input (i.e., in the sizes of $I$, $J$ and $J_s$), even when $\mathcal{M}$ is recursive, and there are exponentially many routes. This representation is complete in the sense that it “embeds” every route for $J_s$. Another main result in this dissertation is that SPIDER can
compute *one route* fast for the selected data if there is one, and can produce another route if necessary. This algorithm is also complete, and runs in polynomial time in the size of the input.

In computing routes, SPIDER adopts a *lazy* approach. Routes are computed only when requested by the user, thus *lazily*, by examining only the source instance $I$, the target instance $J$, the set of selected data $J_s$, and the schema mapping $\mathcal{M}$. SPIDER’s algorithms for computing routes do not rely on any additional information, besides $I$, $J$, $J_s$ and $\mathcal{M}$.

The decision to develop a lazy approach (as opposed to an eager approach) for computing routes in SPIDER was driven by two main considerations. First, we wanted SPIDER to *readily* work in conjunction with existing data exchange and integration systems based on schema mappings, such as Clio and RDA. Indeed, our prototype implementation of SPIDER has been seamlessly deployed on top of the Clio schema mappings system, *without* any changes to Clio’s mapping execution engine. Adopting an eager approach would have required modifications to Clio’s mapping execution engine, so that extra information, helpful in subsequently determining provenance, could be forwarded along with the data to the target. Furthermore, the lazy approach for computing provenance over schema mappings has not been studied in the literature before. Indeed, as shown in Figure 1.3(c), only the eager approach for computing provenance of data over schema mappings has been studied [51, 53, 96]. To date, SPIDER is the only system which adopts the lazy approach in computing provenance of data over schema mappings. Prior to the development of SPIDER, a lazy approach for computing provenance has been studied by Cui et al. [37, 38] in the context of SQL queries. However, our notion of routes is more general compared to the type of provenance studied by Cui et al. [37, 38], and their approach cannot be directly applied to compute routes over schema mappings, as we shall explain in Section 3.6.2.
We have implemented a prototype of SPIDER in conjunction with the Clio schema mapping system. Our implementation uses Clio’s language for specifying schema mappings, which is based on an extension of tgds and egds to handle the hierarchical model [77, 100], in addition to the relational model. Hence, our current implementation of SPIDER handles schema mappings between any combination of relational and XML schemas. We emphasize that although our current prototype is build around Clio’s schema mappings language, SPIDER can be easily deployed on any data exchange or integration system that uses a similar mapping formalism, since it adopts a lazy approach in computing routes.

An empirical evaluation has been conducted to evaluate the feasibility of SPIDER’s algorithms for computing routes. The experimental results show that in general, computing one route can be achieved at interactive speeds. Our results also show that in realistic scenarios where the instances $I$ and $J$ are small, or the number of selected target tuple in $J_s$ is small, computing all routes can also be achieved at interactive speeds. In general, however, we have observed that the algorithm for computing all routes can perform much worse, compared to the algorithm for computing one route. This is explained by the fact that the latter algorithm stops immediately after the first route is discovered, whereas the former is exhaustive. Hence, although it is possible to compute all routes, the ability to compute one route fast, and exploit the “thinking time” of the user in examining that route to generate alternative routes, as needed, is valuable in general.
1.3 A Schema Integration System

Section 1.2 describes how provenance enables data interoperability by facilitating a user’s understanding of the data and mappings used in data exchange and data integration scenarios. Another important aspect that contributes to data interoperability is the design of the integrated schema in data integration scenarios. The integrated schema enables data interoperability in that it provides a standard representation for the source databases, thus offering a way to deal with the heterogeneity in these databases.

The problem of designing an integrated schema based on a set of overlapping source schemas is called schema integration. Schema integration is a long-standing research problem [14, 26, 70, 78, 79, 87] and continues to be a challenge in practice. All the previous approaches require a substantial amount of human feedback during the integration process. Furthermore, the outcome of these approaches is only one integrated schema. In general, however, there can be multiple possible schemas that integrate the data in different ways and each may be valuable in a given scenario. In some of the previous approaches, some of these choices appear implicitly as part of the design process, while interacting with the user. However, there is no principled approach towards the enumeration of these choices. Furthermore, none of the previous approaches (with the exception of the very recent work of [79]) study the problem of automatically generating mappings between the sources and the integrated schema.

This part of the dissertation describes a schema integration method and system [30, 31] that provide: 1) adaptive enumeration of multiple integrated schemas, and 2) easy-to-use capabilities for refining the enumerated schemas via user interaction. Our system takes as input
two or more source schemas (describing data in a common domain) together with a set of correspondences that relate pairs of elements in these schemas. Correspondences signify “semantically equivalent” elements in two schemas. They can be user-specified or discovered through schema matching techniques [80]. Given such input (source schemas and correspondences), the system generates many possible meaningful design choices for the integrated schema. At the same time, it also generates schema mappings that specify how data in each of the source schemas is to be transformed under the integrated schema, for each choice of integrated schema.

A high-level overview of how a user can operate the system is schematically illustrated in Figure 1.4, for the case of two source schemas. As an initial step, each source schema, with its constraints and nesting, is recast into a higher-level graph of concepts. Intuitively, a concept represents one category of data (an entity type) that can exist according to a schema. The subsequent integration method operates at the higher level of abstraction offered by the concept graphs, which abstract away the physical details of relational or XML schemas, thus facilitating a user’s understanding of the integration task. Next, matching concepts in different
graphs are identified by taking into account correspondences between their attributes. For every pair of matching concepts there are two alternatives: merging them into one integrated concept, or leaving them as separate concepts.

At the core of our method for exploring the integrated schemas that result from the above choices (Steps 2 and 3 in Figure 1.4), there are three main components that we have developed. First, we have developed an algorithm for generating one integrated schema, given a fixed choice of which matching concepts to merge. This schema preserves, in a precise sense, all the attributes and relationships of the source schemas. At the same time, the algorithm generates schema mappings that relate the source schemas with the integrated schema and have precise information-preserving properties [70].

Second, we have developed an enumeration algorithm which explores the space of candidate integrated schemas via an enumeration procedure that takes into account all possible merging choices, rather than a fixed one. An essential feature of this algorithm is that it avoids exploring different configurations that yield the same integrated schema. This duplication-free algorithm makes use, as a subroutine, of a polynomial-delay algorithm by Creignou and Hébrard [36] for the enumeration of all Boolean vectors satisfying a set of Horn clauses. Polynomial-delay [58] means that the delay between generating any two consecutive outputs (satisfying assignments in this case) is bounded by a polynomial in the size of the input. In a precise sense, this is the best one can hope for when the number of outputs is exponential in the size of the input.

The duplicate-free enumeration algorithm provides the basis for more efficient and more directed ways of exploring the space of candidate schemas. We have developed one such
directed method (the third component of our system) which allows a user to interactively and selectively explore the space of integrated schemas by expressing constraints on how the concepts should be merged. These constraints are expressed based on the user’s domain knowledge and are incorporated during enumeration adaptively, on the fly, as schemas are being enumerated. Such an adaptive and selective enumeration procedure that can quickly converge to a final integrated schema is very desirable, since the space of candidate schemas may be large in general, thus making the full enumeration unfeasible from a user’s point of view. Although constraints are added primarily for semantic reasons (to incorporate the domain knowledge of the user), they also have the benefit of greatly reducing the space of candidate schemas. Each constraint can cut the space by as much as half.

We have implemented our schema integration system on top of Clio. Experimental results with several real-life integration scenarios show that the algorithm for exploring schemas has good performance, in the sense that the time to generate the next schema is low (sub-second for all the experiments). Thus, a user of the system experiences only a small delay before seeing a different schema. Furthermore, the experimental results show how user interaction can easily, and in a principled way, narrow down the set of candidate schemas to a few relevant ones.

1.4 Organization of the Dissertation

Chapter 2 describes DBNotes, the annotation management system for relational databases useful in tracing the provenance of data over SQL queries. Chapter 3 describes SPIDER, the system for tracing the provenance of data over schema mappings. Chapter 4 describes the schema integration system. Chapter 5 concludes with possible directions for future research.
Chapter 2

The DBNotes System

In this chapter, we study the problem of eagerly tracing the provenance of data that has been transformed by mappings expressed as SQL queries. We describe a principled method for eagerly computing provenance over SQL queries [23] that is based on propagating annotations along with the data, as data is being transformed through queries. We have implemented this method in DBNotes [34], an annotation management system for relational databases.

In DBNotes, each attribute value of a tuple in a relation can have zero or more annotations associated with it, and these annotations propagate along with the data, as data is being transformed from the source to the output. In its default behavior, DBNotes propagates annotations based on where data is copied from (i.e., based on provenance). Hence, an immediate application is to use the propagated annotations to systematically trace the provenance of data. If each column of each tuple in a database is annotated with its address, the provenance of data is eagerly propagated along, as data is being transformed through queries. Hence, the origins and the journey of data can be determined by examining the annotations, even when the data
had undergone multiple transformation steps.

Besides the ability to trace data provenance, DBNotes has many other applications. Annotations may be used to describe information about data that would otherwise have not been kept in a database. For example, an error report or remarks about a piece of data may be attached and propagated along to other databases, thus notifying other users of the error or additional information. The quality or security level of a piece of data can also be described in annotations. Since annotations are propagated along as a query is executed, the annotations on the result of a query can be aggregated to determine the quality or degree of sensitivity of the resulting output. The idea of using annotations to describe the security level of various data items or to specify fine-grained access control policies is not new and can be found in various forms in the literature [39, 57, 71].

This chapter describes the three annotation propagation schemes of DBNotes, each useful for different purposes. They correspond to the default, default-all, and custom propagation schemes. The default scheme propagates annotations based on where data is copied from. If an output piece of data \(d'\) is copied from an input piece of data \(d\), then the annotations associated with \(d\) are propagated to \(d'\). A piece of output data \(d'\) is copied from an input piece of data \(d\) if \(d'\) is created from \(d\) according to the syntax and evaluation of the query. Here, a piece of data refers to a column (i.e., an attribute value) of a tuple in a database.

Although the definition of the default propagation scheme corresponds intuitively to how people reason about provenance, the way annotations are propagated is dependent on the way a query is written. As shown in [91], two equivalent queries may propagate annotations differently. (Two queries are equivalent if they return the same output on any database.) For
instance, consider the relations \( R(A, B) \) and \( S(B, C) \). The following two equivalent queries compute the join of \( R \) and \( S \) on the \( B \) attribute.

\[
Q_1: \quad \text{SELECT } r.B \quad \text{FROM } R r, S s \quad \text{WHERE } r.B = s.B \\
Q_2: \quad \text{SELECT } s.B \quad \text{FROM } R r, S s \quad \text{WHERE } r.B = s.B
\]

Intuitively, it is easy to see that \( Q_1 \) propagates the annotations from the \( B \) attribute of \( R \), while \( Q_2 \) propagates annotations from the \( B \) attribute of \( S \). Since one may be interested in obtaining all relevant annotations of a piece of data in the output regardless of how a query may have been written, DBNotes provides an alternative scheme called default-all, which propagates annotations according to where data is copied from in all equivalent formulations of the given query. Unlike the default scheme, two equivalent queries will always propagate annotations in the same way under this scheme. A given query can therefore be rewritten for optimization purposes, without regard to whether the annotation propagation behavior might change.

At first sight, the default-all scheme seems impossible to implement as there are infinitely many equivalent reformulations of a given query. A main result of this dissertation states that it is always possible to find a finite set of equivalent queries, called a query basis, whose annotation propagation behavior is representative of all equivalent queries. Hence, the annotated output of the given query under the default-all scheme can be obtained by running every query in the query basis, and taking the union of resulting tuples and annotations.

In some cases, a user may only be interested in annotations provided by a certain trusted data source. Hence, DBNotes provides a third propagation scheme, called the custom propagation scheme, where the user is free to specify how annotations should be propagated.

The three propagation schemes have been implemented by extending a fragment of
SQL that corresponds to conjunctive queries with union [3]. This extension is called pSQL. A pSQL query is essentially an SQL query extended with a PROPAGATE clause that would propagate annotations according to one of the schemes described above as data is being transformed. The implementation of DBNotes is based on an underlying annotation storage scheme where for each attribute of each relation, there is an additional column that stores the annotations for that attribute. This chapter describes algorithms for translating a given pSQL query into one or more SQL queries against the underlying annotations storage scheme, which correctly retrieve the relevant annotations according to the specified propagation scheme.

This chapter also describes a series of experiments which have been conducted to evaluate the feasibility of DBNotes. The experimental results indicate that on databases where each attribute value has at most one annotation, the execution times of pSQL queries with default scheme are comparable to those of SQL queries. The performance overhead of propagating annotations was at most 40%. Since the propagated annotations offer the benefit of being able to quickly determine the provenance of any piece of output data, we believe this is an acceptable overhead in general. This result is particularly encouraging, since the use of pSQL with default scheme on databases where each value has at most one annotation is sufficient for tracing data provenance purposes.

The performance of a pSQL query with default-all scheme is worse compared to the performance of the same query under the default scheme. This is explained by the fact that the relevant annotations are obtained by executing each query in the query basis and unioning the resulting annotations. Hence, the performance of the default-all scheme degrades proportionally with the size of the query basis, and there is an additional overhead of unioning the annotations.
Therefore, a default-all pSQL query whose query basis contains \( n \) queries runs at least \( n \) times slower compared with the same query with default clause. While this may be acceptable in special cases (e.g., when the running time of the query is small, or the size of the query basis is small), it is an open question whether the query basis can be optimized in order to achieve the feasibility of the default-all scheme in general.

The experimental results also indicate that the performance of pSQL queries starts to degrade on databases where each attribute value has at least one annotation. This suggests that perhaps a different annotation storage scheme might be better suited in such scenarios. We discuss two possible alternative storage schemes in Section 2.6.1. Trade-offs between different annotation storage schemes have been investigated in [5, 50].

### 2.1 pSQL

This section focuses on a fragment of SQL that corresponds to conjunctive queries with union [3], also known as the Select-Project-Join-Union fragment of SQL. This fragment of SQL is extended with a `PROPAGATE` clause, to allow users to specify how annotations should be propagated.

**Definition 2.1.1** A pSQL query is a query of the form \( Q_1 \ UNION \ldots \ UNION Q_k, k > 0 \), where each \( Q_i, i \in [1, k] \), is a pSQL query fragment of the form shown below:

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad \text{selectlist} \\
\text{FROM} & \quad \text{fromlist} \\
\text{WHERE} & \quad \text{wherelist} \\
\text{PROPAGATE} & \quad \text{DEFAULT | DEFAULT-ALL} | \\
& \quad r_1.A_1 \text{ TO } B_1, \ldots, r_n.A_n \text{ TO } B_n
\end{align*}
\]

The fromlist of a pSQL query fragment is of the form “\( R_1 r_1, \ldots, R_k r_k \)” where \( r_i \) is
a tuple variable of the corresponding relation $R_i$. The selectlist of a pSQL query fragment is of the form “$r_1.C_1 \text{ AS } D_1, ..., r_m.C_m \text{ AS } D_m$” where $r_i$ is a tuple variable defined in fromlist, $C_i$ is an attribute of the relation that corresponds to $r_i$, and $D_i$ is an attribute name of the output relation. The WHERE clause is optional and the wherelist is a conjunction of one or more equalities between attributes of relations or between attributes of relations and constants. The PROPAGATE clause can be defined with DEFAULT, DEFAULT–ALL, or a list of clauses of the form “$r.A \text{ TO } B$” definitions where $r.A$ denotes an attribute $A$ of the tuple that is bound to $r$ and $B$ is an attribute among the $D_j$s.

The SQL query that corresponds to a pSQL query $Q$ is the SQL query that results when all PROPAGATE clauses in $Q$ have been removed. The meaning of a pSQL query is similar to that of its corresponding SQL query except that annotations are also propagated to each emitted tuple according to the specification given in the PROPAGATE clauses.

We note at this point that we assume set semantics for pSQL, as it can be observed from Definition 2.1.1. Furthermore, other set operators (such as intersection or set difference) and aggregate functions are not allowed to appear in a pSQL query. Section 2.6.2 describes how bag semantics may be handled, as well as how pSQL queries can be extended to a larger fragment of SQL where aggregates are also allowed.

**Example 2.1.1** Consider three databases SWISS-PROT (a protein database), PIR (another protein database), and Genbank (a gene database). Each of these databases is modeled as a relation. The schemas and an instance of each relation are shown at the top of Figure 2.1. An annotation, shown in braces, is placed on every column of every tuple. Each annotation can be interpreted as the address of the value in the corresponding column of the tuple. An example of a pSQL
query with the default propagation scheme is shown below.

\[
Q_1 = \begin{aligned}
\text{SELECT} & \ \text{DISTINCT} \ s.\text{ID} \ \text{AS ID,} \ s.\text{Desc} \ \text{AS Desc} \\
\text{FROM} & \ \text{SWISS-PROT} \ s \\
\text{WHERE} & \ s.\text{ID} = "q229" \\
\text{PROPAGATE} & \ \text{DEFAULT}
\end{aligned}
\]

Intuitively, the default scheme specified in \( Q_1 \) propagates annotations of data according to where data is copied from. The result of \( Q_1 \) executed against the relation SWISS-PROT is shown in Figure 2.1. The annotation \( a_3 \) is attached to the value q229 in the output since q229 is copied from the \texttt{ID} attribute of the second tuple in SWISS-PROT. Likewise, \( a_4 \) in the output is propagated from the annotation of the \texttt{Desc} attribute of the second tuple in SWISS-PROT.

While the default scheme is a natural scheme for propagating annotations, this scheme is not robust in that two equivalent queries that return the same output may not propagate the
Example 2.1.2 Consider two equivalent SQL queries \( Q' \) and \( Q'' \) (two queries are equivalent if they produce the same result on every database).

\[
Q' = \text{SELECT DISTINCT } p.ID \text{ AS ID, } p.Name \text{ AS Name} \\
\text{FROM PIR } p, \text{ MappingTable } m \\
\text{WHERE } p.ID = m.pir
\]

\[
Q'' = \text{SELECT DISTINCT } m.pir \text{ AS ID, } p.Name \text{ AS Name} \\
\text{FROM PIR } p, \text{ MappingTable } m \\
\text{WHERE } p.ID = m.pir
\]

The results of running \( Q' \) and \( Q'' \) under the default propagation scheme are shown below.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>p332</td>
<td>{a7} AB {a_8}</td>
</tr>
<tr>
<td>p916</td>
<td>{a9} AB {a_{10}}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>p332</td>
<td>{a_{17}} AB {a_8}</td>
</tr>
<tr>
<td>p916</td>
<td>{a_{21}} AB {a_{10}}</td>
</tr>
</tbody>
</table>

For \( Q' \), the annotations for the ID column are from the PIR table while for \( Q'' \), the annotations for the ID column are from the MappingTable.

While it is likely that a user will realize that \( Q' \) will generate a different annotated outcome from \( Q'' \) in general, the situation is not so straightforward for more complex queries. The above example motivates the need for a propagation scheme that is invariant under equivalent queries. A user should be able to retrieve all relevant annotations about a piece of output data regardless of how the query is written, if desired. The default-all propagation scheme propagates annotations according to where data is copied from among all equivalent formulations of the given query. Hence the annotated outcome is the same for equivalent queries under this scheme. In case a user prefers to retrieve annotations from one source over another, the user is also free to specify how annotations should propagate in the custom scheme.
Example 2.1.3 The queries $Q_2$ and $Q_3$ are examples of pSQL queries with the default-all and custom propagation schemes respectively.

$$Q_2 = \text{SELECT DISTINCT } p.\text{ID AS ID}, p.\text{Name AS Name}$$
$$\text{FROM PIR } p$$
$$\text{PROPAGATE DEFAULT-ALL}$$

$$Q_3 = \text{SELECT DISTINCT } g.\text{ID AS ID}, g.\text{Desc AS Desc}$$
$$\text{FROM Genbank } g$$
$$\text{PROPAGATE } g.\text{ID TO ID}, g.\text{Desc TO ID}$$

The results of $Q_2$ and $Q_3$ are shown at the bottom of Figure 2.1. The query $Q_2$ retrieves all tuples from the PIR table under the default-all propagation scheme. Since the following query is equivalent to $Q_2$,

$$Q'_2 = \text{SELECT DISTINCT } p.\text{ID AS ID}, p.\text{Name AS Name}$$
$$\text{FROM PIR } p, \text{PIR } q$$
$$\text{WHERE } p.\text{Name} = q.\text{Name}$$

annotations of proteins with the same name are combined together. As a consequence, the protein with name AB has both annotations $a_8$ and $a_{10}$. Intuitively, the annotations obtained in the result of a default-all pSQL query fragment $Q$ are the combined annotations of results from all equivalent queries of $Q$. In the custom scheme of $Q_3$, annotations are propagated according to the given user specification (i.e., $g.\text{ID TO ID}, g.\text{Desc TO ID}$). A clause “$g.\text{ID TO ID}$” states that the annotations associated with the value of the ID attribute of the tuple that is currently bound to $g$ should propagate to the ID attribute of the output tuple. Similarly, the annotations associated to the value of the Desc attribute of the tuple that is currently bound to $g$ should propagate to the ID attribute of the output tuple.

Some Terminology A cell (or location) is a triple $(R, t, i)$ which denotes the $i$th column of the tuple $t$ in relation $R$. We sometimes use the attribute name at position $i$ instead of the position $i$. A cell may also be denoted as a pair $(t, i)$, whenever the relation name $R$ is understood from
the context. Let \( \mathcal{L} \) denote the set of all strings. Each cell \( c \) in a database is associated with a set of annotations \( \{a_1, \ldots, a_k\} \) where each \( a_i, i \in [1, k] \), is an element in \( \mathcal{L} \). We say that each \( a_i, i \in [1, k] \), is an annotation attached to \( c \). The notation \( \mathcal{A}(r, t, i) \) is used to denote the set of all annotations attached to the cell \( (R, t, i) \). Similarly, \( \mathcal{A}(t, i) \) denotes the set of all annotations attached to the cell \( (t, i) \) in the context where the relation \( R \) is clear.

**Example 2.1.4** Figure 2.1 shows several examples of annotated relations. The value \( z131 \) in SWISS-PROT is the value at cell \( \text{(SWISS-PROT, (z131, AB), ID)} \) which denotes the ID column of tuple \( (z131, AB) \) in the SWISS-PROT relation. The attribute names in the tuple \( (z131, AB) \) have been omitted. The annotation \( \{a_1\} \) is the set of annotations associated with this cell. Hence, \( \mathcal{A} \text{(SWISS-PROT, (z131, AB), ID)} \) is \( \{a_1\} \). In the result of \( Q_2 \), \( \mathcal{A}((p332, AB), \text{Name}) \) is \( \{a_8, a_{10}\} \).

**Containment vs. Annotation-Containment** Two pSQL queries \( Q \) and \( Q' \) are equivalent, denoted as \( Q = Q' \), if for every database \( D \), it is the case that \( Q(D) = Q'(D) \). The query \( Q \) is contained in \( Q' \), denoted as \( Q \subseteq Q' \), if for every database \( D \), it is the case that \( Q(D) \subseteq Q'(D) \). Two pSQL queries \( Q \) and \( Q' \) are annotation-equivalent [91], denoted as \( Q =_a Q' \), if \( Q \) and \( Q' \) produce the same annotated output on all databases. More precisely, this means that for every database \( D \), \( Q(D) \) is equal to \( Q'(D) \) and the set of annotations \( \mathcal{A}(Q(D), t, i) \) is identical to \( \mathcal{A}(Q'(D), t, i) \) for every output cell \( (t, i) \) in \( Q(D) \). A pSQL query \( Q \) is annotation-contained [91] in \( Q' \), denoted as \( Q \subseteq_a Q' \), if for every database \( D \), it is the case that \( Q(D) \subseteq Q'(D) \) and furthermore, for every output cell \( (t, i) \) in \( Q(D) \), it is the case that \( \mathcal{A}(Q(D), t, i) \subseteq \mathcal{A}(Q'(D), t, i) \).
Example 2.1.5 The queries $Q'$ and $Q''$ in Example 2.1.2 are equivalent. However, they are not annotation-equivalent since different annotations are associated with the results. Consider the following query $Q$:

$$Q = \text{SELECT DISTINCT } g.\text{ID AS ID, } g.\text{Desc AS Desc}
\text{FROM Genbank } g
\text{PROPAGATE } g.\text{ID TO ID, } g.\text{Desc TO ID, } g.\text{ID TO Desc}$$

The query $Q_3$ of Example 2.1.3 is annotation-contained in $Q$ since they are equivalent and the annotations associated with each cell in the result of $Q_3$ is contained in the set of annotations associated with the corresponding cell in the result of $Q$. Intuitively, $Q_3$ is annotation-contained in $Q$ because they are equivalent and the ID attribute in the selectlist of both queries receive the same annotations from $g.\text{ID}$ and $g.\text{Desc}$. Furthermore, the Desc attribute in the selectlist of $Q$ receives annotations from $g.\text{ID}$.

2.1.1 The Custom Propagation Scheme

The user of DBNotes is allowed the flexibility to specify custom propagation schemes using a PROPAGATE clause of the form “$r_1.A_1 \text{ TO } B_1$, ..., $r_n.A_n \text{ TO } B_n$”. The queries $Q_3$ of Example 2.1.3 and $Q$ of Example 2.1.5 are examples of pSQL queries with custom propagation scheme. The semantics of a pSQL query fragment $Q$ with custom propagation scheme is as follows. For every binding $\mu$ of tuple variables to tuples in the respective relations according to the fromlist of $Q$ such that the conditions in the wherelist are satisfied, emit an output tuple $t$ according to the selectlist. For every clause “$r_i.A_i \text{ TO } B_i$” specified in the PROPAGATE clause, add the set of annotations at the cell $(r_i, A_i)$ to the set of annotations (initially empty) at the output cell $(t, B_i)$. Finally, duplicate output tuples are merged, since set semantics are assumed.
Suppose \( t_1, \ldots, t_k \) are the emitted tuples and \( s_1, \ldots, s_m \) are the tuples that result when duplicate output tuples have been merged. Then, for every output cell \( (s, B) \), the set \( \mathcal{A}(s, B) \) is obtained as \( \bigcup_{t_j=s, j \in [1,k]} \mathcal{A}(t_j, B) \).

**Example 2.1.6** To illustrate the effect of removing duplicate output tuples and merging annotations of duplicate tuples, consider the query below:

```sql
SELECT DISTINCT Name AS Name
FROM PIR
PROPAGATE DEFAULT
```

The result of executing the above query will merge the annotations \( a_8 \) and \( a_{10} \) of the Name values of the first and second tuple in PIR. Hence the final output is a single tuple (AB) with annotations \( \{a_8, a_{10}\} \).

As another example, the query \( Q_3 \) of Example 2.1.3 has a custom propagation scheme where annotations on both ID and Desc columns of each tuple propagate to the ID column of the output tuple. As a consequence, the ID column of every output tuple is the union of annotations associated with the ID and Desc columns of the corresponding tuple in Genbank.

### 2.1.2 The Default Propagation Scheme

If `PROPAGATE DEFAULT` is used in a pSQL query fragment, the set of annotations of a piece of output data consists of all the annotations associated with the cells where that piece of data is copied from in the source.

The semantics of a pSQL query fragment \( Q \) with the default propagation scheme is as follows. For every binding of tuple variables to tuples in the respective relations according
to the fromlist of \( Q \) such that the conditions in the wherelist are satisfied, emit an output tuple \( t \) according to the selectlist, as well as the corresponding sets of annotations for every cell in \( t \). Since every value of an output cell \( c' \) in \( t \) is generated from some value of an input cell \( c \) according to the current bindings, the set of annotations attached to \( c \) is also attached to \( c' \).

Finally, duplicate output tuples are merged together. Suppose \( t_1, \ldots, t_k \) are the emitted tuples, and \( s_1, \ldots, s_m \) are the tuples that result when duplicate output tuples have been merged. That is, for every output cell \((s, B)\), we have \( A(s, B) = \bigcup_{t_j = s, j \in [1,k]} A(t_j, B) \).

**Example 2.1.7** Consider the following pSQL query where each fragment uses the default propagation scheme.

```
SELECT DISTINCT Desc AS Desc
FROM SWISS-PROT
PROPAGATE DEFAULT
UNION
SELECT DISTINCT Desc AS Desc
FROM Genbank
PROPAGATE DEFAULT
```

The result is:

<table>
<thead>
<tr>
<th>Desc</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>{a_2, a_{12}}</td>
</tr>
<tr>
<td>CC</td>
<td>{a_4, a_{14}}</td>
</tr>
<tr>
<td>ED</td>
<td>{a_6}</td>
</tr>
</tbody>
</table>

The first subquery emits an output tuple “AB” with annotations \{a_2\} and the second subquery emits the same output tuple “AB” but with annotations \{a_{12}\}. The merged result of these two tuples is a single output tuple “AB” with annotations \{a_2, a_{12}\}. This explains the first output tuple in the result. A similar reasoning applies to the rest of the output tuples.

A pSQL query fragment with default propagation scheme can easily be translated into a pSQL query fragment with custom propagation scheme. For example, the query \( Q_1 \) of Example 2.1.1 can be rewritten into a pSQL query with custom scheme where the propagate clause is replaced by “PROPAGATE \( s.\text{ID} \) TO ID, \( s.\text{Desc} \) TO Desc” since the ID value and Desc value of an output tuple are copied from \( s.\text{ID} \) and \( s.\text{Desc} \), respectively.
2.1.3 The Default-All Propagation Scheme

A pSQL query with the default propagation scheme is, essentially, an SQL query with annotations propagated based on where a value is retrieved from according to the syntax of the query. Two pSQL queries with default propagation scheme which are equivalent but not annotation-equivalent have been described in Example 2.1.2.

This motivates the third propagation scheme of DBNotes, called the default-all scheme, where the annotation propagation behavior of a pSQL query is invariant to the syntax of the query. A pSQL query $Q$ with default-all propagation scheme propagates annotations according to the default propagation behavior of all equivalent formulations of $Q$. The resulting tuples that are generated by all equivalent queries of $Q$ according to the default scheme are then merged together. Despite the fact that there are infinitely many equivalent formulations of $Q$, we show that it is always possible to compute the desired result by examining only a finite number of pSQL queries. Such a finite set of queries is called a query-basis of $Q$.

**Definition 2.1.2** Let $Q$ denote a pSQL query with default-all propagation scheme. Let $SQL(Q)$ denote the SQL query that corresponds to $Q$ and let $E(SQL(Q))$ denote the set of all pSQL queries $Q'$ under the default propagation scheme such that $SQL(Q')$ is equivalent to $SQL(Q)$. A query basis of $Q$, denoted as $B(Q)$, is a finite set of pSQL queries such that

$$\bigcup_{q \in B(Q)} q =_{a} \bigcup_{q \in E(SQL(Q))} q$$

The next section describes an algorithm that finds a query basis for a pSQL query with default-all propagation scheme. The size of the query basis that the algorithm returns is always polynomial in the size of $Q$. (The size of a query basis is the sum of sizes of each
pSQL query fragment in the query basis. The size of each pSQL query fragment is the sum of the number of attributes in the selectlist, the number of relations in the fromlist and the number of attributes appearing in the whereclist.)

2.2 Generating a Query Basis

The algorithm for computing a query basis for a pSQL query with default-all propagation scheme proceeds by first generating a representative query of $Q$, called $Q_0$. (This is Step 1 of the algorithm Generate-Query-Basis shown below.) Intuitively, a representative query of $Q$ is a query that is equivalent to $Q$ and for every attribute $B$ that is equal or transitively equal to an attribute $A$ in the selectlist of $Q$, the annotations of $B$ are propagated to $A$. More precisely, if $A$ is among the selectlist and we have $A = B$ and $D = B$ in the whereclist of $Q$, then the propagatelist will contain the propagate clauses “$A \text{ TO } A$”, “$B \text{ TO } A$” and “$D \text{ TO } A$”.

From $Q_0$, a finite number of auxiliary queries are also generated and these queries, together with $Q_0$, form a query basis of $Q$. (This is Step 2 of the algorithm.) Each auxiliary query is equivalent to $Q$ but may propagate additional annotations to the output that are not propagated by $Q_0$. In other words, every output value may contain additional annotations from attributes of other relations which contain identical values. Intuitively, only a finite number of auxiliary queries are needed because only one auxiliary query needs to be generated for each attribute of a relation that contributes annotations to the output. In the rest of the discussion, we restrict our language to be pSQL query fragments. In other words, a query basis of $Q$, denoted as $B(Q)$, is a finite set of pSQL query fragments such that $\bigcup_{q \in B(Q)} q = a \bigcup_{q \in E(SQL(Q))} q$, 39
where $E(\text{SQL}(Q))$ denotes the set of pSQL query fragments $Q'$ such that $\text{SQL}(Q')$ is equivalent to $\text{SQL}(Q)$.

The algorithm for generating a query basis of a pSQL query fragment with default-all propagation scheme is shown below. Section 2.3 describes an extension of this algorithm to handle pSQL queries (i.e., union of pSQL query fragments).

---

**Algorithm Generate-Query-Basis**

**Input:** A pSQL query fragment $Q$ with default-all propagation scheme.

**Output:** A query basis of $Q$, $B(Q)$.

Let $Q$ be a pSQL query fragment of the form shown in Definition 2.1.1 with \texttt{PROPAGATE DEFAULT-ALL} clause.

1. **Generate $Q_0$, the representative query of $Q$.**

   Generate a query $Q_0$ that is identical to $Q$ except that the propagation scheme of $Q$ is replaced with the following propagation scheme:

   For every attribute “$r.A \text{ AS } C$” in the \texttt{selectlist}, add “$r.A \text{ TO } C$” to the \texttt{PROPAGATE} clause.

   For every attribute “$r.A \text{ AS } C$” in the \texttt{selectlist} and every attribute $s.B$ that is equal to $r.A$ or transitively equal to $r.A$ according to the \texttt{wherelist}, add “$s.B \text{ TO } C$” to the \texttt{PROPAGATE} clause.

   (The effect is that all attributes that are equal to an attribute $C$ in the \texttt{selectlist} have their annotations propagated to $C$.)

2. **Generate auxiliary queries of $Q_0$.**

   Initialize $B(Q)$ to the empty set. Add $Q_0$ to $B(Q)$. For every attribute “$r.A \text{ AS } C$” in the \texttt{selectlist} of $Q_0$ and every “$s.B \text{ TO } D$” in the \texttt{PROPAGATE} clause of $Q_0$ where $C = D$, do the following:

   Create a query $Q'$ that is identical to $Q_0$. Assume that $s$ is a tuple variable for relation $S$. Add
“S s’” to the fromlist of $Q’$ where $s’$ is a tuple variable that does not occur in $Q’$. Add “$s’.B = s.B$” to the where list of $Q’$ and “$s’.B \text{ TO } C’” to the PROPAGATE clause of $Q’$. (The auxiliary query $Q’$ is equivalent to $Q$ but may carry additional annotations to the output.)

3. Return $B(Q)$.

Example 2.2.1 Consider the three databases, SWISS-PROT, PIR, and Genbank along with a Mapping Table that contains the correspondences between identifiers of genes and proteins in the three databases in Figure 2.1. Such mapping tables commonly occur in integrating many sources with overlapping information [60]. Consider the following query $Q$ that integrates information from SWISS-PROT and PIR.

$$
Q = \text{SELECT DISTINCT } t.\text{swissprot AS ID, } p.\text{Name AS Name, } s.\text{Desc AS Desc}
\text{FROM Mapping Table } t, \text{ SWISS-PROT } s, \text{ PIR } p
\text{WHERE } t.\text{swissprot} = s.\text{ID AND } t.\text{pir} = p.\text{ID}
\text{PROPAGATE DEFAULT-ALL}
$$

Step 1 of the algorithm generates the following representative query $Q_0$:

$$
Q_0 = \text{SELECT DISTINCT } t.\text{swissprot AS ID, } p.\text{Name AS Name, } s.\text{Desc AS Desc}
\text{FROM Mapping Table } t, \text{ SWISS-PROT } s, \text{ PIR } p
\text{WHERE } t.\text{swissprot} = s.\text{ID AND } t.\text{pir} = p.\text{ID}
\text{PROPAGATE } t.\text{swissprot TO ID, } s.\text{ID TO ID, } p.\text{Name TO Name, } s.\text{Desc TO Desc}
$$

The annotations of $t.\text{swissprot}$ and $s.\text{ID}$ will propagate to the output ID column according to $Q_0$. Step 2 of the algorithm generates four auxiliary queries. The first query is shown below and the rest are shown in Figure 2.2.

$$
Q_1 = \text{SELECT DISTINCT } t.\text{swissprot AS ID, } p.\text{Name AS Name, } s.\text{Desc AS Desc}
\text{FROM Mapping Table } t, \text{ SWISS-PROT } s, \text{ PIR } p, \text{ Mapping Table } t'
\text{WHERE } t.\text{swissprot} = s.\text{ID AND } t.\text{pir} = p.\text{ID AND } t'.\text{swissprot} = t.\text{swissprot}
\text{PROPAGATE } t.\text{swissprot TO ID, } s.\text{ID TO ID, } p.\text{Name TO Name, } s.\text{Desc TO Desc, } t'.\text{swissprot TO ID}
$$
\[ Q_2 = \text{SELECT DISTINCT } t\text{.swissprot AS ID, } p\text{.Name AS Name, } s\text{.Desc AS Desc} \]
\[ \text{FROM Mapping\_Table } t, \text{ SWISS-PROT } s, \text{ PIR } p, \text{ SWISS-PROT } s' \]
\[ \text{WHERE } t\text{.swissprot = } s\text{.ID AND } t\text{.pir = } p\text{.ID AND } s'\text{.ID = } s\text{.ID} \]
\[ \text{PROPAGATE } t\text{.swissprot TO ID, } s\text{.ID TO ID, } p\text{.Name TO Name, } s\text{.Desc TO Desc, } s'\text{.ID TO ID} \]

\[ Q_3 = \text{SELECT DISTINCT } t\text{.swissprot AS ID, } p\text{.Name AS Name, } s\text{.Desc AS Desc} \]
\[ \text{FROM Mapping\_Table } t, \text{ SWISS-PROT } s, \text{ PIR } p, \text{ SWISS-PROT } s' \]
\[ \text{WHERE } t\text{.swissprot = } s\text{.ID AND } t\text{.pir = } p\text{.ID AND } s'\text{.Desc = } s\text{.Desc} \]
\[ \text{PROPAGATE } t\text{.swissprot TO ID, } s\text{.ID TO ID, } p\text{.Name TO Name, } s\text{.Desc TO Desc, } s'\text{.Desc TO Desc} \]

\[ Q_4 = \text{SELECT DISTINCT } t\text{.swissprot AS ID, } p\text{.Name AS Name, } s\text{.Desc AS Desc} \]
\[ \text{FROM Mapping\_Table } t, \text{ SWISS-PROT } s, \text{ PIR } p, \text{ PIR } p' \]
\[ \text{WHERE } t\text{.swissprot = } s\text{.ID AND } t\text{.pir = } p\text{.ID AND } p'\text{.Name = } p\text{.Name} \]
\[ \text{PROPAGATE } t\text{.swissprot TO ID, } s\text{.ID TO ID, } p\text{.Name TO Name, } s\text{.Desc TO Desc, } p'\text{.Name TO Name} \]

Figure 2.2: Some of the auxiliary queries generated by Step 2 of Generate-Query-Basis on Example 2.2.1.

The query \( Q_1 \) differs from \( Q_0 \) only in the additional highlighted terms shown in \( Q_1 \).

There is an extra relation, condition and propagation in the FROM, WHERE and PROPAGATE clauses respectively. It is easy to verify that the SQL queries of \( Q_0 \) and \( Q_1 \) are equivalent. There is a homomorphism \( h \) from the tuple variables of \( Q_1 \) to those of \( Q_0 \) such that \( h \) maps the fromlist of \( Q_1 \) to a subset of the fromlist of \( Q_0 \) and the conditions in the wherelist of \( Q_0 \) imply the conditions in the wherelist of \( Q_1 \) under \( h \). Furthermore, \( h \) maps the selectlist of \( Q_1 \) to the selectlist of \( Q_0 \). There is also a homomorphism in the reverse direction. Similarly, \( Q_2, Q_3, \) and \( Q_4 \) of Figure 2.2 are each equivalent to \( Q_0 \).

Intuitively, the representative query \( Q_0 \) propagates annotations according to where data is copied from and also where data could have been equivalently copied from. The reason
why $Q_0$ is generated becomes clearer if we represent $Q$ in conjunctive query-like notation, which is used in the rest of the discussion, for ease of exposition. In conjunctive query-like notation, a query $Q$ is represented as

$$H(\bar{x}) : -S_1(\bar{y}_1), ..., S_n(\bar{y}_n), \text{equalities}.$$ 

where $\bar{x}, \bar{y}_i, i \in [1, n]$, denote vectors of variables and every variable in $\bar{x}$ occurs in $\bar{y}_i$ for some $i \in [1, n]$ and \textit{equalities} is a list of zero or more $y = c$ clauses, where $y$ is a variable that occurs amongst $\bar{y}_i$s and $c$ is a constant. The variables in $\bar{x}$ are called \textit{distinguished variables}. Each subgoal corresponds to a relation in the \textit{fromlist} of $Q$. The equalities between attributes in the \textit{wherelist} of $Q$ are represented by using the same variable in the respective positions of relations in the conjunctive query-like representation of $Q$. An equality between an attribute and constant is written out as \textit{equalities}. The head of the query $H(\bar{x})$ represents the \textit{selectlist} of $Q$. It is easy to see that there is always a unique translation of the SQL query which corresponds to $Q$ into the conjunctive query-like representation. The notation $C(Q)$ is used to denote the conjunctive query-like representation of the SQL query that corresponds to $Q$. For example, the conjunctive query-like representation $C(Q_0)$ of $Q_0$ from Example 2.2.1 is as follows:

$$H_0(x, y, z) : -\text{Mapping_Table}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y).$$

Similar to the semantics of pSQL queries with the default propagation scheme, annotations are propagated according to where data is copied from for such queries [91] by tracing the occurrence of distinguished variables in the query. For example, by tracing the occurrence of the variable $x$ in the query $H_0$, it can be concluded that the annotations in the first column
of an output tuple \( t \) are obtained from the annotations of the second column of a tuple in Mapping Table and the first column of a tuple in SWISS-PROT that created \( t \). A similar argument applies to the variables \( y \) and \( z \) in \( H_0 \). Hence, the representative query \( Q_0 \) of Example 2.2.1 is annotation-equivalent to \( C(Q_0) \). Note that a query may not be annotation equivalent to its conjunctive query-like representation, in general. To illustrate, consider two relations \( R(A, B) \) and \( S(B, C) \) and the following query \( Q' \) whose conjunctive query-like representation \( C(Q') \) is \( H_0(y) :\!:: R(x, y), S(y, z) \).

\[
Q': \begin{align*}
& \text{SELECT } r.B \\
& \text{FROM } R r, S s \\
& \text{WHERE } r.B = s.B
\end{align*}
\]

It is easy to see that only the annotations from \( R.B \) propagate to the result according to \( Q' \), whereas according to \( C(Q') \), annotations from both \( R.B \) and \( S.B \) propagate to the result. Therefore, \( Q' \) and \( C(Q') \) are not annotation equivalent.

### 2.2.1 Correctness of Generate-Query-Basis

This section shows that given a query \( Q \), the algorithm Generate-Query-Basis(\( Q \)) correctly generates \( B(\mathcal{Q}) \), the query basis of \( Q \). First, Proposition 2.2.1 shows that the representative query \( Q_0 \) generated by the algorithm is annotation-equivalent to its conjunctive query-like representation, \( C(Q_0) \). With this result, we shall show that the conjunctive query-like representation of each query generated by the Generate-Query-Basis algorithm is annotation contained in \( \bigcup_{\mathcal{Q} \in B(\mathcal{Q})} \mathcal{Q} \), the union of all queries in \( B(\mathcal{Q}) \) (Proposition 2.2.2). Intuitively, Propositions 2.2.1 and Proposition 2.2.2 show that translating into the internal conjunctive query notation does not create any additional annotations propagating to the output besides the ones propagated by the
pSQL queries in the basis. This allows us to reason in terms of the conjunctive query-like representation, as opposed to pSQL queries, which is more convenient in the proofs. Lemma 2.2.1 makes use of the two propositions to show that every query that is equivalent to $Q$ is annotation contained in $\bigcup_{q \in B(Q)} q$. Finally, the main result of this section is given in Theorem 2.2.2 which states that the algorithm Generate-Query-Basis correctly generates a query basis $B(Q)$ for the input query $Q$. The result follows immediately from Lemma 2.2.1 and the fact that each query in the basis is equivalent to $Q$.

Next, we first show that the representative query $Q_0$ generated by the algorithm is annotation equivalent to its conjunctive query-like representation.

**Proposition 2.2.1** The representative query $Q_0$ that is generated by Generate-Query-Basis($Q$) is annotation-equivalent to its conjunctive query-like representation, $C(Q_0)$.

**Proof.** Obviously, the query $Q_0$ is equivalent to $C(Q_0)$ since there is a subgoal $S$ in $C(Q_0)$ for every relation $S$ in the fromlist of $Q_0$ and vice versa, there is an equality condition $e$ in $C(Q_0)$ for every equality condition $e$ the wherelist of $Q_0$ and vice versa, and the head of $C(Q_0)$ produces the same attributes as the selectlist of $Q_0$. Next, we shall show that $Q_0$ and $C(Q_0)$ are annotation-equivalent, by showing that for every database $D$ and every output cell $(t, i)$ of $Q_0(D)$, the set of annotations $A(Q_0(D), t, i)$ is equal to $A(C(Q_0)(D), t, i)$. We show that if a cell $c'$ in the source $D$ corresponds to a cell $c$ in the output of $C(Q_0)(D)$, then the annotations at $c'$ are part of the annotations at $c$ according to $Q_0$ and $D$. The converse also holds.

According to the semantics of conjunctive queries with annotation propagation stated in [91] (Appendix A.1), the set of annotations associated with an output cell $c$ is the union of the
sets of annotations associated to each source cell $c'$ that corresponds to $c$. A cell $(s, i)$ in $D$ corresponds to $(t, j)$ in $C(Q_0)(D)$, where $C(Q_0)$ is of the form $H(\bar{x}) : -S_1(\bar{y}_1), ..., S_n(\bar{y}_n)$, equalities, if there exists $k \in [1, n]$ such that $\bar{y}_k[i] = \bar{x}[j]$ and there exists a valuation $\varphi$ from $C(Q_0)$ into $D$ such that $H(\varphi(\bar{x})) = t$, $S_k(\varphi(\bar{y}_k)) = s$ and the equalities are satisfied.

Suppose there is such a valuation $\varphi$ for $C(Q_0)$ as stated above. Then there is a valuation $\varphi'$ for $Q_0$ that produces $t$. The valuation $\varphi'$ is such that $\varphi'(r) = S(\varphi(\bar{y}))$ where $r$ is a tuple variable in $Q_0$ and $S(\bar{y})$ is the corresponding subgoal in $C(Q_0)$ which represents the relation that $r$ ranges over. So $\varphi'(u) = s$ for the tuple variable $u$ in $Q_0$ which ranges over the relation $S_k$ and the output tuple is $t$ under $\varphi'$ according to $Q_0$. Since $\bar{y}_k[i] = \bar{x}[j]$ in $C(Q_0)$, it must be that the attribute at position $i$ of $S_k$ (call it $A$) is equal to the attribute at position $j$ in the selectlist of $Q_0$ (call it $B$) or transitively equal to $B$. Hence there must be a clause “PROPAGATE $u.A$ TO $B$” in the propagate clause of $Q_0$. Therefore under the valuation $\varphi'$, the annotations at $(s, i)$ are part of the annotations at $(t, j)$ according to $Q_0$ and $D$.

For the converse, suppose there is a valuation $\varphi$ for $Q_0$ and $D$ such that the annotations at $(s, i)$ are part of the annotations at $(t, j)$ according to $Q_0$ and $D$ with $\varphi$. So $\varphi(u) = s$ for some tuple variable $u$ in $Q_0$ and the output tuple is $t$ under $\varphi$. Clearly, there must also be a valuation $\varphi'$ for $C(Q_0)$ and $D$ that produces $t$. The valuation $\varphi'$ is such that $S(\varphi'(\bar{y})) = \varphi(r)$ where $r$ is a tuple variable in $Q_0$, $S(\bar{y})$ is a subgoal in $C(Q_0)$ and $S$ is the relation that $r$ ranges over. So there exists a subgoal $S_k(\bar{y}_k)$ in $C(Q_0)$ for some $k \in [1, n]$ such that $S_k(\varphi'(\bar{y}_k)) = s = \varphi(u)$. Let the $i$th attribute of $s$ be $A$ and the $j$th attribute of the output tuple $t$ be $B$. Hence there must be a “PROPAGATE $u.A$ TO $B$” clause in $Q_0$ and “$v.C$ AS $B$” is in the selectlist for some tuple variable $v$ and attribute $C$. According to the Generate-Query-Basis algorithm, this means
that either \( u.A \) is equal to \( v.C \) or transitively equal to \( v.C \). Hence, in \( C(Q_0) \), it must be that 
\[
\tilde{y}_k[i] = \tilde{x}[j].
\]
It follows that \( \tilde{y}_k[i] = \tilde{x}[j] \), \( H(\varphi'(\tilde{x})) = t \), \( S_k(\varphi'(\tilde{y}_k)) = s \) and the equalities are satisfied under \( \varphi' \). Hence, \( (s, i) \) corresponds to \( (t, j) \) according to \( C(Q_0) \) and \( D \) with the valuation \( \varphi' \).

In Step 2, the algorithm generates one query for every position in the body where a distinguished variable occurs in \( H_0 \). For example, the following four auxiliary queries, in conjunctive query notation, are generated based on \( H_0 \). They are annotation-equivalent to the pSQL query fragments \( Q_1, \ldots, Q_4 \) shown in Example 2.2.1 and Figure 2.2, respectively.

\[
\begin{align*}
H_1(x, y, z) &::= \text{Mapping}_{\text{Table}}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y). \quad \text{Mapping}_{\text{Table}}(w_1, x, w_2, w_3). \\
H_2(x, y, z) &::= \text{Mapping}_{\text{Table}}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y). \quad \text{SWISS-PROT}(x, w_1). \\
H_3(x, y, z) &::= \text{Mapping}_{\text{Table}}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y). \quad \text{SWISS-PROT}(w_1, z). \\
H_4(x, y, z) &::= \text{Mapping}_{\text{Table}}(w, x, u, v), \text{SWISS-PROT}(x, z), \text{PIR}(u, y), \text{PIR}(w_1, y).
\end{align*}
\]

Although in our example it happened that the auxiliary pSQL queries are each annotation equivalent to their conjunctive query-like representation, this may not be true in general. The next proposition guarantees, however, that translating into the conjunctive query notation does not incur in any additional annotations which are not propagated by the pSQL queries in the basis.

**Proposition 2.2.2** For every query \( Q' \in B(Q) \) where \( B(Q) \) is the result of \( \text{Generate-Query-Basis}(Q) \), \( C(Q') \) is annotation-contained in \( \bigcup_{q \in B(Q)} q \).

**Proof.** First, \( C(Q_0) \) is annotation-contained in \( \bigcup_{q \in B(Q)} q \) since \( Q_0 \in B(Q) \) and \( C(Q_0) \) is annotation-equivalent to \( Q_0 \) according to Proposition 2.2.1.
Let $Q'$ denote a query in $\mathcal{B}(Q)$ and $Q'$ is not $Q_0$. That is, $Q'$ is one of the auxiliary queries. Let $C(Q')$ be of the form \( H(\bar{x}) : -S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n), \) equalities. Given any database $D$, let $(s, i)$ be a cell in $D$ which corresponds to a cell $(t, j)$ in $C(Q')(D)$ on a valuation $\varphi$. So $S_k(\varphi(\bar{y}_k)) = s$ for some $k \in [1, n]$ and $H(\varphi(\bar{x})) = t$ and $\bar{y}_k[i] = \bar{x}[j]$. There is also a valuation $\varphi'$ for $Q'$ and $D$ which produces $t$. The valuation $\varphi'$ is such that $\varphi'(r) = S(\varphi(\bar{y}))$ where $r$ is a tuple variable in $Q'$ and $S(\bar{y})$ is the corresponding subgoal in $C(Q')$ which represents the relation that $r$ ranges over in $Q'$. So $\varphi'(r_1) = s$ for the tuple variable $r_1$ in $Q'$ which ranges over the relation $S_k$ and the output tuple is $t$ under $\varphi'$ according to $Q'$. We show next that for every annotation propagated by $Q'$, there is a query in $\mathcal{B}(Q)$ that would propagate the annotation in the same way.

Suppose $S_k(\bar{y}_k)$ is a subgoal among the subgoals of $C(Q_0)$ where $Q_0$ is the representative query generated by Step 1 of the algorithm Generate-Query-Basis. (Recall that $C(Q')$ differs from $C(Q_0)$ in that it has an additional subgoal added by Step 2 of the algorithm.) Since $\bar{y}_k[i] = \bar{x}[j]$ and $S_k(\bar{y}_k)$ is a subgoal among the subgoals of $C(Q_0)$, it must be that the attribute at position $i$ of $S_k$ (call it $B$) is equal to the attribute at position $j$ in the selectlist of $Q'$ (call it $A$) or transitively equal to $A$. Hence, there must be a clause \"PROPAGATE $r_1.B$ TO $A$\" in the propagate clause of $Q_0$ (and hence $Q'$). Therefore under the valuation $\varphi'$, the annotations at $(s, i)$ are part of the annotations at $(t, j)$ according to $Q'$ and $D$.

Suppose $S_k(\bar{y}_k)$ is not a subgoal among the subgoals of $C(Q_0)$. That is, $S_k(\bar{y}_k)$ is the subgoal that corresponds to the extra relation in the fromlist, added by Step 2 of the algorithm Generate-Query-Basis. Let the attribute at the $i$th position of $S_k$ be $B$. By Step 2 of the algorithm, it must be that the condition \"$r_1.B = r_2.B$\" is the added condition in the
where list for some tuple variable \( r_2 \) that ranges over a second \( S_k \) relation in \( Q' \) and \( "r_1.B \ TO C" \) is the added propagate clause of \( Q' \) for some output attribute \( C \) in the select list. Let the attribute at the \( j \)th position of the output be \( A \). If \( C \) is the same as \( A \), then the annotations at \((s, i)\) are part of the annotations at \((t, j)\) according to \( Q' \) and \( D \) under the valuation \( \varphi' \). Suppose \( C \) is not equal to \( A \). Since \( "r_1.B = r_2.B" \) and \( \tilde{y}_k[i] = \tilde{x}[j] \) in \( C(Q') \), it must be that \( r_2.B \) is equal or transitively equal to \( A \). (Therefore \( Q_0 \) must contain \( "r_2.B \ TO A" \) in the propagate clause.) Hence there must be a query \( q \) in \( B(Q) \) which is identical to \( Q' \) except that \( "r_1.B \ TO A" \) is in the propagate clause instead of \( "r_1.B \ TO C" \). Therefore under the valuation \( \varphi' \), the annotations at \((s, i)\) are part of the annotations at \((t, j)\) according to \( q \) and \( D \). 

Each auxiliary query carries annotations to the output that may have been missed by the representative query of \( Q \). Lemma 2.2.1 below makes use of Propositions 2.2.1 and 2.2.2 to show that none of the necessary annotations is in fact missed by the representative and auxiliary queries combined. Concretely, the lemma shows that every query which is equivalent to \( Q \) is annotation contained in \( \bigcup_{q \in B(Q)} q \). It follows that \( \bigcup_{q \in E(SQL(Q))} q \subseteq \bigcup_{q \in B(Q)} q \), where recall that \( E(SQL(Q)) \) denotes the set of all pSQL queries \( Q' \) with default clause such that the SQL queries of \( Q \) and \( Q' \) are equivalent. The converse (i.e., \( \bigcup_{q \in B(Q)} q \subseteq \bigcup_{q \in E(SQL(Q))} q \)) follows easily from the fact that the SQL queries which correspond to each of the queries in \( B(Q) \) are equivalent to \( Q \). Hence, the set of pSQL query fragments in \( B(Q) \) generated by the algorithm is a query basis for \( Q \). This result is stated in Theorem 2.2.2 given after the proof of the lemma.

**Lemma 2.2.1** Let \( B(Q) \) denote the result produced by the algorithm Generate-Query-Basis(\( Q \)), where \( Q \) is a pSQL query fragment, and let \( Q' \) denote a pSQL query fragment under the default
propagation scheme. If $Q'$ is equivalent to $Q$, then $Q'$ is annotation-contained in $\bigcup_{q \in B(Q)} q$.

**Proof.** The representative query $Q_0$ that is generated at Step 1 of the algorithm is annotation-equivalent to the conjunctive query representation of the SQL query that corresponds to $Q$, $C(Q)$ (Proposition 2.2.1). It can be also verified that $Q' \subseteq_a C(Q')$. Since $C(Q)$ and $C(Q')$ are equivalent queries, the minimal queries of $C(Q)$ and $C(Q')$ are identical up to variable renaming. For convenience, we shall assume that the minimal queries are identical in the form shown below. We also assume that there are no equalities between variables and constants, for convenience. (A minimal query is a query in which no subquery, one that has less subgoals or joins, is equivalent to it.)

$$C(Q): H(\bar{x}) \leftarrow \text{minpart}, \text{rest1}.$$  
$$C(Q'): H(\bar{x}) \leftarrow \text{minpart}, \text{rest2}.$$  

The subgoals denoted by minpart are the subgoals in the minimal query of $C(Q)$ or $C(Q')$ and rest1 and rest2 denote the rest of the subgoals in $C(Q)$ and $C(Q')$, respectively. The proof makes use of an earlier result in [91] extended for unions of conjunctive queries. Given a conjunctive query $Q$, the notation $Q[0]$ denotes the head of $Q$, the notation $Q[i], i > 0$, denotes the $i$th subgoal of $Q$, and $\text{var}(Q[i])$ denotes the list of variables of the $i$th subgoal of $Q$.

**Fact 1 ([92], Appendix A.3)** Given two unions of conjunctive queries $Q = \bigcup_{i=1}^{m} Q_i$ and $Q' = \bigcup_{j=1}^{n} Q'_j$, $Q \subseteq_a Q'$ if and only if for every $Q_r$ where $r \in [1, m]$, every variable $x$, every $i$, and every $p$ such that $x$ that occurs at both the $i$th position of $\text{var}(Q_r[0])$ and the $j$th position of $\text{var}(Q'_r[p])$, there exists a homomorphism $h$ from $Q'_s$ (for some $s \in [1, n]$) to $Q_r$ such that

1. $h$ maps the body of $Q'_s$ into the body of $Q_r$ and the head of $Q'_s$ to the head of $Q_r$, and
2. the variable that occurs at the $j$th position of the $q$th subgoal of $Q'_s$ (i.e., $\text{var}(Q'_s[q])[j]$) is identical to the variable at the $i$th position of the head of $Q'_s$ (i.e., $\text{var}(Q'_s[0])[i]$), where $Q'_s[q]$ is a pre-image of $Q_r[p]$ under $h$. That is, for some subgoal $q$, $\text{var}(Q'_s[q])[j] = \text{var}(Q'_s[0])[i]$ and $h(Q'_s[q]) = Q_r[p]$.

We show next that for every distinguished variable $x$ at the $i$th position in the head of $C(Q'_0)$ and its occurrence at the $j$th position of the $p$th subgoal $S(\bar{u})$ (i.e., the $j$th variable of $\bar{u}$ is $x$) in the body of $C(Q'_0)$, there is a generated query $Q_g$ in $B(Q)$ and a homomorphism $h : C(Q_g) \rightarrow C(Q'_0)$ that satisfies the conditions (1) and (2) stated in the fact. By Fact 1, it follows that $C(Q'_0) \subseteq_a C(Q_g)$. By Proposition 2.2.2, $C(Q_g) \subseteq_a \bigcup_{q \in B(Q)} q$. Therefore $C(Q'_0) \subseteq_a \bigcup_{q \in B(Q)} q$. Since $Q' \subseteq_a C(Q'_0)$ and $C(Q'_0) \subseteq_a \bigcup_{q \in B(Q)} q$, it follows that $Q' \subseteq_a \bigcup_{q \in B(Q)} q$, which was to be shown.

Let $x$ be a distinguished variable at the $i$th position in the head of $C(Q'_0)$ and suppose $x$ occurs at the $j$th position of the $p$th subgoal $S(\bar{u})$ of $C(Q'_0)$.

Case 1. If $S(\bar{u})$ is among the subgoals in the minpart of $C(Q'_0)$, then it must also be among the subgoals in the minpart of $C(Q)$. Hence the algorithm Generate-Query-Basis would have generated one or more queries whose combined effect is the query $C(Q_g)$, shown below,

\[
H(\bar{x}) :- \text{minpart, rest1}, S(\bar{w}_1, x, \bar{w}_2).
\]

The variable $x$ occurs at the $j$th position in the subgoal $S(\bar{w}_1, x, \bar{w}_2)$ and $\bar{w}_1$ and $\bar{w}_2$ are vectors of distinct variables that do not occur in $C(Q)$. This corresponds to Step 2 of the algorithm where a new relation $S$ is added to the FROM clause. (Note that a clause “$B \text{ TO } A$” is also added to the PROPAGATE clause to simulate the effect of $x$ propagating
annotations to the output. We assume that $x$ occurs under the attribute $A$ in the output and $B$ is the attribute name of $x$ in $S$ in the named perspective. If $x$ occurs under another attribute $D$ in the output of $C(Q_g)$, there will be another query generated by Step 2 of the algorithm that propagates the annotations of $B$ to $D$. Hence there is possibly more than one pSQL query whose combined annotation propagation effect equals that of $C(Q_g)$. It is easy to see that there is a homomorphism from $C(Q_g)$ to $C(Q')$ with the desired properties required by the fact shown above. The homomorphism is obtained by extending the homomorphism $h' : C(Q) \rightarrow C(Q')$ which must exist, since $C(Q)$ and $C(Q')$ are equivalent. The homomorphism $h'$ is extended to $h''$ by mapping the $i$th variable in $\bar{w}_1$ to the corresponding $i$th variable in $\bar{u}$ and the $i$th variable in $\bar{w}_2$ to the $(j+i)$th variable in $\bar{u}$ (this is possible since $\bar{w}_1$ and $\bar{w}_2$ are distinct variables). Clearly, $h''$ satisfies the conditions required by Fact 1.

Case 2. If $S(\bar{u})$ are among the subgoals in rest2 of $C(Q')$, we first show that a subgoal $S(\bar{w}')$, where the $j$th variable of $\bar{w}'$ is $x$, must also occur among subgoals in the minpart of $Q'$. With this, a similar argument presented before shows that there must be a homomorphism from a query $C(Q_g)$ to $C(Q')$ with the desired conditions required by Fact 1, which was to be shown.

We show next that if $S(\bar{u})$ are among the subgoals in rest2 of $C(Q')$, there must exist such a subgoal $S(\bar{w}')$ among the minpart of $C(Q')$ where the $j$th variable of $\bar{w}'$ is $x$. Since there is a homomorphism $g$ from $C(Q')$ to the minimal query of $C(Q')$ and $g(x) = x$ (since $x$ is a distinguished variable), this implies that there must be a subgoal $S(...x...)$ among the subgoals in the minpart of $C(Q')$ such that $x$ occurs at the $j$th position of this subgoal. Therefore, it follows that $S(\bar{w}')$ exists.
Theorem 2.2.2 Let $Q$ be a pSQL query fragment with default-all propagation scheme. The algorithm $\text{Generate-Query-Basis}(Q)$ returns a query basis of $Q$.

Proof. Let $E(Q)$ denote the set of pSQL query fragments $q$ under the default propagation scheme such that the SQL query that corresponds to $q$ is equivalent to that of $Q$ (i.e., $\text{SQL}(q) = \text{SQL}(Q)$). Let $B(Q)$ denote the result of running the algorithm $\text{Generate-Query-Basis}$ on $Q$. By Lemma 2.2.1, we have that $\bigcup_{q \in E(Q)} q \subseteq a \bigcup_{q \in B(Q)} q$.

For the converse, recall that the SQL queries of $Q$ and $Q_0$ are equivalent, by Step 1 of the algorithm. Furthermore, by Step 2 or the algorithm we have that the SQL queries of $Q_0$ and $Q_i$ are equivalent, for each auxiliary query $Q_i$ generated by the algorithm. (An explanation of this fact was also given at the end of Example 2.2.1.) Therefore, the SQL queries of the query $Q$, its representative $Q_0$ and all its auxiliary queries are equivalent. Hence, $B(Q) \subseteq E(Q)$. It follows that $\bigcup_{q \in B(Q)} q \subseteq a \bigcup_{q \in E(Q)} q$ and hence the result.

The next proposition shows that the size of a query basis is polynomial in the size of $Q$. The size of a query basis is the sum of sizes of each pSQL query fragment in the query basis. The size of each pSQL query fragment is the sum of the number of attributes in the selectlist, the number of relations in the fromlist and the number of attributes appearing in the wherelist. This result shows that the result of executing a query basis is polynomial in the size of the database (data complexity).

Proposition 2.2.3 Given a pSQL query fragment $Q$ with default-all propagation scheme, the number of queries returned by $\text{Generate-Query-Basis}(Q)$ is polynomial in the size of $Q$. Furthermore, each query in $\text{Generate-Query-Basis}(Q)$ is polynomial in the size of $Q$. 53
Proof. Let $s$, $f$, and $w$ denote the number of clauses in the selectlist, number of relations in the fromlist, and number of equalities in the wherelist of $Q$, respectively. The size of $Q$ consists in the sum of the number of attributes in the selectlist, the number of relations in the fromlist and the number of attributes appearing in the wherelist, that is $|Q|$ is at most $s + f + 2 * w$.

One representative query $Q_0$ is generated by the algorithm. The size of the propagate list of $Q_0$ is at most $s + s * 2 * w$. (In the worst case, every attribute in the wherelist propagates to every attribute in the selectlist.) The number of auxiliary queries generated is therefore at most $s * (s + s * 2 * w)$ which is $|selectlist| * |propagatelist|$. Hence, the total number of queries in Generate-Query-Basis($Q$) is at most $1 + s * (s + s * 2 * w)$.

The size of the selectlist, fromlist, wherelist, and propagatelist of $Q_0$ is $s$, $f$, $w$, and at most $s + s * 2 * w$, respectively. The size of each auxiliary query is thus at most $s + (f + 1) + (w + 2) + (s + s * 2 * w + 1)$ since one additional relation, one condition, and one propagate clause is added to $Q_0$.

An optimization The auxiliary pSQL queries overlap significantly in the propagate clauses. This behavior can be observed in Figure 2.2, where the auxiliary queries shown differ only in the last (highlighted) propagation. Intuitively, the non-highlighted propagations in the auxiliary queries are unnecessary, because these propagations are identical to the propagations of the representative query $Q_0$. Hence, in the optimized implementation of Generate-Query-Basis, these non-highlighted propagations are not generated in the auxiliary queries. The original implementation of the algorithm Generate-Query-Basis is referred to as the unoptimized implementation.
2.3 Generating a Query Basis for General pSQL Queries

This section describes an extension of the Generate-Query-Basis algorithm to handle pSQL queries (i.e., union of pSQL query fragments).

To generate a query basis for a pSQL query $Q = Q_1 \cup \ldots \cup Q_l$ where each $Q_i, i \in [1, l]$, is a pSQL query fragment with default-all propagation scheme, the algorithm Generate-Query-Basis described in Section 2.2 is modified to the following algorithm, called Generate-Containment-Basis. Step 1 of Generate-Containment-Basis remains the same as in Generate-Query-Basis. The algorithm Generate-Containment-Basis differs from Generate-Query-Basis in Step 2, where for each pSQL query fragment $Q_i (1 \leq i \leq l)$, the set of all queries that are contained in $Q_i$ are generated and added to the auxiliary queries of $Q_i$. A consequence of this effect is that a query that is identical to $Q_i$ but with an additional relation $R$ that does not occur in $Q_i$ is considered as a query contained in $Q_i$. Hence, annotations from $R$ may propagate to the output. In contrast, Step 2 of Generate-Query-Basis generates a set of auxiliary pSQL query fragments that are each equivalent to $Q_i$. The algorithm Generate-Containment-Basis computes a query basis for a given pSQL query. The algorithm is described below and it is illustrated with an example next.

**Algorithm** Generate-Containment-Basis

**Input:** A pSQL query $Q = Q_1 \cup \ldots \cup Q_l$ with default-all propagation scheme.

**Output:** A query basis of $Q$, $B(Q)$.

Let $Q$ be a pSQL query of the form $Q = Q_1 \cup \ldots \cup Q_l$ where each $Q_i, i \in [1, l]$, is a pSQL query fragment of the form shown in Definition 2.1.1 with PROPAGATE DEFAULT-ALL clause. For each $Q_i, i \in [1, l]$,
we execute the following two steps.

1. **Generate $Q^i_0$, the representative query of $Q_i$.**
   
   Generate a query $Q^i_0$ that is identical to $Q_i$ except that the propagation scheme of $Q_i$ is replaced with the following propagation scheme:
   
   For every attribute “$r.A \ AS C$” in the selectlist, add “$r.A TO C$” to the PROPAGATE clause.
   
   For every attribute “$r.A \ AS C$” in the selectlist and every attribute $s.B$ that is equal to $r.A$ or transitively equal to $r.A$ according to the where list, add “$s.B TO C$” to the PROPAGATE clause.
   
   (The effect is that all attributes that are equal to an attribute $C$ in the selectlist have their annotations propagated to $C$.)

2. **Generate auxiliary queries of $Q^i_0$.**
   
   Initialize $B(Q_i)$ to the empty set. Add $Q^i_0$ to $B(Q_i)$.
   
   For every clause “$s.A \ AS B$” in the selectlist of $Q^i_0$, for every relation $R$ in the database and every attribute $C$ in the relation schema of $R$:
   
   Create a query $Q'$ that is identical to $Q^i_0$. Add “$R r$” to the fromlist of $Q'$ where $r$ is a tuple variable that does not occur in $Q'$. Add the condition “$r.C = s.A$” to the where list of $Q'$ and the propagate clause “$r.C TO B$” to the propagatelist of $Q'$. Add $Q'$ to $B(Q_i)$.
   
   (The query $Q'$ is contained in $Q_i$ but may propagate additional annotations. Furthermore, we have that $\bigcup_{Q \in B(Q_i)} q$ is equivalent to $Q_i$.)

Return $B(Q_1) \cup \ldots \cup B(Q_m)$.

---

**Example 2.3.1** Consider a database consisting of the following relations: Emp(name, dept), Dept(did, budget), Project(proj, mgr). Consider following query $Q$ which selects employees who belong to some department.
Two of the queries generated by Step 2 of the algorithm Generate-Containment-Basis on $Q$ are shown below:

$$Q_1 = \text{SELECT DISTINCT } e.\text{name AS Name}$$
$$\text{FROM } \text{Emp } e, \text{Dept } d, \text{Project } p$$
$$\text{WHERE } e.\text{dept} = d.\text{did AND } p.\text{proj} = e.\text{name}$$
$$\text{PROPAGATE } e.\text{name TO Name, p.proj TO Name}$$

$$Q_2 = \text{SELECT DISTINCT } e.\text{name AS Name}$$
$$\text{FROM } \text{Emp } e, \text{Dept } d, \text{Project } p$$
$$\text{WHERE } e.\text{dept} = d.\text{did AND } p.\text{mgr} = e.\text{name}$$
$$\text{PROPAGATE } e.\text{name TO Name, p.mgr TO Name}$$

The highlighted parts of $Q_1$ and $Q_2$ denote the additional relation, condition and propagate clauses added to the representative query $Q_0$ by Step 2 of the algorithm. Observe that $Q_1$ and $Q_2$ are queries that are contained in $Q$ (but $Q_0 \cup Q_1 \cup Q_2$ is equivalent to $Q$).

Furthermore, $Q_1$ propagates annotations on projects to the result and $Q_2$ propagates annotations from the names of managers to the result. Arguably, $Q_1$ should not have been generated since the annotations for projects are irrelevant for names of employees. However, in the absence of additional semantic information, omitting $Q_1$ might lead to missing important annotations. The query $Q_2$, propagates the annotations for a manager to an employee name and this is desired since the manager and the employee have the same name (and are therefore referring to the same entity).

A few observations follow from Example 2.3.1. First, since the language now allows for union, the query basis for $Q$ contains more queries than the query basis generated by Generate-Query-Basis for the same query $Q$. This is because in the case of Generate-Query-Basis, only pSQL query fragments that are equivalent to $Q$ need to be considered. In contrast,
all pSQL queries that are equivalent to $Q$ have to be considered by the Generate-Containment-Basis algorithm. The example also suggests that a more refined method of generating a query basis is needed. Namely, one should only generate an auxiliary query if it propagates relevant annotations. In the above example, $Q_2$ is desired, while $Q_1$ is not. To generate only auxiliary queries that propagate relevant annotations, one would require the knowledge of semantically equivalent attributes in a database to be kept in the system. Queries are then generated by adding the extra relation and equating only semantically equivalent attributes. In what follows, we shall assume that both $Q_1$ and $Q_2$ are generated. Also, observe that by equating “$r.C = s.A$” in Step 2, it is assumed that all attributes have the same type.

2.3.1 Correctness of Generate-Containment-Basis

The algorithm Generate-Containment-Basis takes as input $Q = Q_1 \cup \ldots \cup Q_l$ and generates as output a query basis $B(Q)$ which is $B(Q_1) \cup \ldots \cup B(Q_l)$. The following proposition is similar to Proposition 2.2.2, adapted for queries generated by the algorithm Generate-Containment-Basis. The Lemma 2.3.1 and Theorem 2.3.2 given after the proposition extend parallel results from Section 2.2 (i.e., Lemma 2.2.1 and Theorem 2.2.2) to pSQL queries.

**Proposition 2.3.1** For every query $Q'$ in the result of Generate-Containment-Basis($Q$) (denoted as $B(Q)$), $C(Q')$ is annotation-contained in $\bigcup_{q \in B(Q)} q$.

**Proof.** Let $Q = Q_1 \cup \ldots \cup Q_l$ and let $Q_0$ denote the representative query generated by Step 1 of the algorithm for query $Q_i$. Clearly $C(Q_0)$ is annotation-contained in $\bigcup_{q \in B(Q_i)} q$, since $Q_0 \in B(Q_i)$ and $C(Q_0)$ is annotation-equivalent to $Q_0$ according to Proposition 2.2.1.
Let $Q'$ denote a query in $B(Q)$ and $Q'$ is not $Q^i_0$ for every $i \in [1, l]$. That is, $Q'$ is one of the auxiliary queries, generated by Step 2 of the algorithm. Let $C(Q')$ be of the form $H(\bar{x}) : -S_1(\bar{y}_1), \ldots, S_n(\bar{y}_n)$, equalities. Given any database $D$, let $(s, i)$ be a cell in $D$ which corresponds to a cell $(t, j)$ in $C(Q')(D)$ on a valuation $\varphi$. So $S_k(\varphi(\bar{y}_k)) = s$ for some $k \in [1, n]$ and $H(\varphi(\bar{x})) = t$ and $\bar{y}_k[i] = \bar{x}[j]$. There is also a valuation $\varphi'$ for $Q'$ and $D$ which produces $t$. The valuation $\varphi'$ is such that $\varphi'(r) = S(\varphi(\bar{y}))$ where $r$ is a tuple variable in $Q'$ and $S(\bar{y})$ is the corresponding subgoal in $C(Q')$ which represents the relation that $r$ ranges over in $Q'$. So $\varphi'(r_1) = s$ for some tuple variable $r_1$ in $Q'$ and the output tuple is $t$ under $\varphi'$ according to $Q'$. We show next that for every annotation propagated by $Q'$, there is a query in $B(Q)$ that would propagate the annotation in the same way.

Suppose $Q'$ is in $B(Q_i)$ for some $i \in [1, l]$ and $S_k(\bar{y}_k)$ is a subgoal among the subgoals of $C(Q^i_0)$ where $Q^i_0$ is the representative query generated by Step 1 of the algorithm Generate-Containment-Basis. (Recall that $C(Q')$ differs from $C(Q^i_0)$ in that it has an additional subgoal added by Step 2 of the algorithm.) Since $\bar{y}_k[i] = \bar{x}[j]$ and $S_k(\bar{y}_k)$ is a subgoal among the subgoals of $C(Q^i_0)$, it must be that the attribute at position $i$ of $S_k$ (call it $B$) is equal to the attribute at position $j$ in the selectlist of $Q'$ (call it $A$) or transitively equal to $A$. Hence, there must be a clause "PROPAGATE $r_1.B$ TO $A$" in the propagate clause of $Q^i_0$ (and hence $Q'$). Therefore under the valuation $\varphi'$, the annotations at $(s, i)$ are part of the annotations at $(t, j)$ according to $Q'$ and $D$.

Suppose $S_k(\bar{y}_k)$ is not a subgoal among the subgoals of $C(Q^i_0)$. That is, $S_k(\bar{y}_k)$ is the subgoal that corresponds to the extra relation in the fromlist, added by Step 2 of algorithm Generate-Containment-Basis. Let the attribute at the $i$th position of $S_k$ be $C$. Since $\bar{y}_k[i] = \bar{x}[j]$
and by Step 2 of the algorithm, it must be that the condition “\(r_1.C = r_2.A\)” is the added condition in the \textit{wherelist} for some tuple variable \(r_2\) that ranges over a relation in \(Q'\). The clause “\(r_1.C \bowtie B\)” is the added propagate clause of \(Q'\) for some output attribute \(B\) in the \textit{selectlist}. (Hence “\(r_2.A \bowtie B\)” is among the \textit{selectlist} of \(Q'\).) Let the attribute at the \(j\)th position of the output be \(F\). If \(B\) is the same as \(F\), then the annotations at \((s, i)\) are part of the annotations at \((t, j)\) according to \(Q'\) and \(D\) under the valuation \(\varphi'\). Suppose \(B\) is not equal to \(F\). Since “\(r_1.C = r_2.A\)” and \(y_k[i] = x[j]\) in \(C(Q')\), it must be that \(r_2.A\) is equal or transitively equal to the output attribute \(F\) (according to \(Q^i_0\)). In other words, let “\(r_3.E \bowtie F\)” be the select clause for \(F\) (which is among the \textit{selectlist} of \(Q^i_0\)) where \(r_3\) is a tuple variable in \(Q^i_0\) and \(E\) is an attribute. We have \(r_3.E\) is either equal or transitively equal to \(r_2.A\) according to the \textit{wherelist} of \(Q^i_0\). Hence the following query \(Q''\) from \(Q^i_0\) will be generated: \(Q''\) has an extra relation “\(S_k r_1\)” in the \textit{fromlist}, the added condition “\(r_1.C = r_3.E\)” in the \textit{wherelist} and the added propagate clause “\(r_1.C \bowtie F\)”.

Since under \(\varphi'\), we have \(r_1.C = r_2.A\) and \(r_2.A\) is equal or transitively equal to \(r_3.E\), it follows that \(r_1.C = r_3.E\). Therefore the valuation \(\varphi'\) is also a valuation for \(Q''\). Hence the annotations at \((s, i)\) are part of the annotations at \((t, j)\) according to \(Q''\) and \(D\).

\[\Box\]

Lemma 2.3.1 shown below is used in the proof of Theorem 2.3.2, which states the correctness of the algorithm \textit{Generate-Containment-Basis}(\(Q\)) in generating a query basis for \(Q\). The proof of the lemma makes use of the following result from [82].

\textbf{Fact 2 ([82])} Let \(Q = \bigcup_{i \in [1, m]} Q_i\) and \(Q' = \bigcup_{i \in [1, n]} Q'_i\) be unions of conjunctive queries. Then \(Q \subseteq Q'\) if and only if for every \(Q_i\), \(i \in [1, m]\), there exists \(Q'_j\), \(j \in [1, n]\), such that \(Q_i \subseteq Q'_j\).
Lemma 2.3.1 Let $\mathcal{B}(Q)$ denote the result produced by the algorithm Generate-Containment-Basis($Q$), where $Q$ is a pSQL query and let $Q'$ denote a pSQL query under the default propagation scheme. If $Q'$ is equivalent to $Q$, then $Q'$ is annotation-contained in $\bigcup_{q \in \mathcal{B}(Q)} q$.

Proof. Let $Q = Q_1 \cup ... \cup Q_l, \bigcup_{q \in \mathcal{B}(Q)} q = q_1 \cup ... \cup q_m$ and let $Q' = Q'_1 \cup ... \cup Q'_n$.

We show next that for every distinguished variable $x$ at the $i$th position in the head of $\mathcal{C}(Q'_f)$ where $f$ in $[1, n]$ and its occurrence at the $j$th position of the $k$th subgoal $S(\bar{u})$ (i.e., the $j$th variable for $\bar{u}$ is $x$) of $\mathcal{C}(Q'_f)$, there is a generated query $q_g \in \mathcal{B}(Q)$ such that there is a homomorphism $h : \mathcal{C}(q_g) \rightarrow \mathcal{C}(Q'_f)$ that satisfies conditions (1) and (2) of Fact 1. Then by Fact 1, we have $\mathcal{C}(Q'_1) \cup ... \cup \mathcal{C}(Q'_n) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} \mathcal{C}(q)$. For every pSQL query fragment $Q$, it is the case that $Q \subseteq_a \mathcal{C}(Q)$. So we have $Q' \subseteq_a \mathcal{C}(Q'_1) \cup ... \cup \mathcal{C}(Q'_n)$ and therefore $Q' \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} \mathcal{C}(q)$. By Proposition 2.3.1, $\bigcup_{q \in \mathcal{B}(Q)} \mathcal{C}(q) \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$. Hence we have $Q' \subseteq_a \bigcup_{q \in \mathcal{B}(Q)} q$, which was to be shown.

Consider a distinguished variable $x$ that occurs at the $i$th position in the head of $\mathcal{C}(Q'_f)$ where $f \in [1, n]$ and at the $j$th position of the $k$th subgoal $S(\bar{u})$ of $\mathcal{C}(Q'_f)$. Hence, $\mathcal{C}(Q'_f)$ has the form shown below, where $x$ occurs at the $i$th position in $H(...)$, $S$ is the $k$th subgoal and $x$ occurs at the $j$th position in $S(...)$.

\[
\mathcal{C}(Q'_f) : H(...x...) := ..., S(\bar{z}_1, x, \bar{z}_2), ...
\]

By Fact 2, since $\mathcal{C}(Q)$ is equivalent to $\mathcal{C}(Q')$, there exists a query $\mathcal{C}(Q_g)$ for some $g \in [1, l]$ such that $\mathcal{C}(Q'_f) \subseteq C(Q_g)$. Consequently, there is a containment mapping $h' : \mathcal{C}(Q_g) \rightarrow \mathcal{C}(Q_f)$. Accordingly, Generate-Containment-Basis would generate a query $q_g$ according to $Q_g$ such that $\mathcal{C}(q_g)$ is identical to $\mathcal{C}(Q_g)$ but has an additional subgoal $S(\bar{w}_1, y, \bar{w}_2)$ where $\bar{w}_1$ and
\( \bar{w}_2 \) are vectors of fresh variables that do not occur elsewhere in \( C(q_g) \). That is, \( C(q_g) \) has the form shown below where \( y \) is the distinguished variable that occurs in the \( i \)th position in the head of \( C(Q_g) \) and the \( j \) position in \( S(\ldots) \).

\[
C(q_g) : H(\ldots y \ldots) :\text{ body of } C(Q_g), S(\bar{w}_1, y, \bar{w}_2).
\]

It is easy to see that there is a homomorphism \( h : C(q_g) \rightarrow C(Q'_g) \) that satisfies conditions (1) and (2) of Fact 1. The homomorphism \( h \) is such that \( h(x) = h'(x) \) for every \( x \in \text{var}(Q_g) \), \( h(\bar{w}_1) = \bar{z}_1 \), and \( h(\bar{w}_2) = \bar{z}_2 \). Clearly, \( h \) is consistent with \( h' \) and is a homomorphism from \( C(q_g) \) to \( C(Q'_g) \). Hence, by Fact 1, \( C(Q'_g) \subseteq_a C(q_g) \).

**Theorem 2.3.2** Given a pSQL query \( Q \), the algorithm Generate-Containment-Basis\( (Q) \) generates a query basis of \( Q \).

**Proof.** Let \( Q \) be a pSQL query \( Q_1 \cup \ldots \cup Q_l \). Let \( Q_R \) denote the union of all queries in \( B(Q) \).

That is, \( Q_R = \bigcup_{q \in B(Q)} q \). Clearly, \( Q_R \) is contained in \( Q \) since each \( B_i(Q_i), i \in [1, l] \), contains a representative query which is equivalent to \( Q_i \) and every other query in \( B_i(Q_i) \) is contained in \( Q_i \). The query \( Q \) is also contained in \( Q_R \) since for every \( Q_i, i \in [1, l] \), \( Q_i \) is equivalent to the representative query of \( Q_i \) in \( Q_R \). Let \( \mathcal{E}(Q) \) denote the set of all equivalent queries of \( Q \) where each query in \( \mathcal{E}(Q) \) propagates using the default scheme. Since \( Q_R \) is equivalent to \( Q \), we have \( Q_R \in \mathcal{E}(Q) \). Hence \( Q_R \subseteq \bigcup_{q \in \mathcal{E}(Q)} q \). By Lemma 2.3.1, \( q \subseteq_a Q_R \) for every query \( q \in \mathcal{E}(Q) \). Therefore \( \bigcup_{q \in \mathcal{E}(Q)} q \subseteq_a Q_R \) and hence \( Q_R =_a \bigcup_{q \in \mathcal{E}(Q)} q \).
2.4 System Architecture

DBNotes is implemented in Java v1.4.2 on top of the Oracle 9i Enterprise Edition Release 9.2.0.1.0 relational database management system (RDBMS). Figure 2.3 illustrates the architecture of DBNotes which consists of two main modules: the translator module and the postprocessor module. The translator module takes as input a pSQL query and returns as output an SQL query (i.e., a union of SPJ queries) which is sent to the RDBMS. The SQL query is then executed by the RDBMS. The tuples that are returned by the RDBMS are sorted in a certain order and sent to the postprocessor module which merges annotations of identical cells of duplicate tuples together in one pass through the sorted tuples.

2.4.1 A Naive Storage Scheme

The annotations are stored in DBNotes using a naive storage scheme, where for every attribute $A$ of a relation scheme $R$ there is an extra column $A_a$ that will be used to store annotations. This new relation with extra columns is denoted as $R'$. For example, a relation $R(A, B)$ will be represented as $R'(A, A_a, B, B_a)$ in the naive storage scheme. Given a tuple $t$ in a relation of $R$, if $\{a_1, ..., a_k\}$ are the annotations associated with the cell $(t, A)$, then there will be $k$ tuples $t_1, ..., t_k$ in $R'$ such that $t_i.A_a = a_i$ for $i \in [1, k]$ and the projection of $t_i$ on the attributes of $R$ equals $t$, for $i \in [1, k]$. For convenience, the relation name $R$ may be sometimes used to
refer to $R'$ As an example, the two instances of $R$ shown below are both valid representations of the tuple $(a \{a_1, a_2\}, b \{b_1\})$.

<table>
<thead>
<tr>
<th></th>
<th>$A_a$</th>
<th>$B_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$a_1$</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>$a_2$</td>
<td>$b_1$</td>
</tr>
</tbody>
</table>

Observe that a query returns the same result regardless of the underlying storage instance used.

**Propagating Provenance** To use DBNotes to automatically propagate provenance along, we first associate each cell with a distinct annotation to denote its address. In what is shown below, $R'$ is defined as a view of an original relation $R$ using internal row identifiers:

```
CREATE VIEW $R'$ AS
SELECT $A$ AS $A_a$, rowid\#$A_a$,
      $B$ AS $B_a$, rowid\#$B_a$
FROM $R$
```

For the above view definition, rowid is an internal row identifier used in many database systems such as Oracle and Postgres. Section 2.4.4 describes how DBNotes makes use of these “address” annotations to automatically trace the provenance of data.

### 2.4.2 The Translator

The translator module takes as input a pSQL query $Q$ and translates $Q$ to an SQL query $Q'$ against the naive storage scheme. A pSQL query with default or default-all propagation scheme is first reformulated into one with a custom propagation scheme. A pSQL query with the custom propagation scheme is reformulated into an SQL query (i.e., a union of SPJ queries). The algorithm for reformulating a pSQL query fragment with default propagation scheme into a pSQL fragment with custom propagation scheme is described briefly at
the end of Section 2.1.2. The methods for reformulating pSQL queries with default-all propagation scheme into a pSQL queries with custom propagation scheme have been described by algorithms Generate-Query-Basis and Generate-Containment-Basis in Sections 2.2 and 2.3, respectively. The algorithm for reformulating a pSQL query with custom propagation scheme into an SQL query is described next.

Algorithm Custom-pSQL-To-SQL

**Input:** A pSQL query fragment \( Q \) with custom propagation scheme.

**Output:** An SQL query \( Q_s \) written against the naive schema.

Let \( Q \) be a pSQL query fragment of the form shown in Definition 2.1.1 with a custom-propagatelist.

1. *Generate intermediate SQL queries.* Each intermediate SQL query retrieves annotations (as much as possible) from the naive schema according to the given query \( Q \).

   Let \( Q_0 \) be a query that is identical to \( Q \) except that it does not have the `PROPAGATE` clause of \( Q \).

   - For each output attribute \( C \) of \( Q \), create an empty bin for \( C \). Denote this bin as \( \text{bin}(C) \). For each propagate clause “\( s.B \text{ TO} C \)” in the custom-propagatelist of \( Q \), add “\( s.B_a \text{ AS} C_a \)” to \( \text{bin}(C) \).

   - Let \( Q \) be the empty set of SQL queries. Repeat until all bins are empty. Let \( Q' \) be a query that is identical to \( Q_0 \). For each output attribute \( C \) of \( Q \), if \( \text{bin}(C) \) is nonempty, remove a clause “\( s.B_a \text{ AS} C_a \)” from \( \text{bin}(C) \) and add it to the `selectlist` of \( Q' \). If \( \text{bin}(C) \) is empty, we add “\( \text{NULL AS} C_a \)” to the `selectlist` of \( Q' \). Add \( Q' \) to \( Q \).

2. *Generate a wrapper SQL query \( Q_s \) for \( Q \).*

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad * \\
\text{FROM} & \quad (Q_1 \text{ UNION} \cdots \text{ UNION } Q_n) \\
\text{ORDER BY} & \quad \text{orderbylist}
\end{align*}
\]
where \( Q = \{Q_1, \ldots, Q_n\} \) and \( \text{orderbylist} \) is the list of all output attributes in the \( \text{selectlist} \) of \( Q \).

The \( \text{orderbylist} \) is required so that the Postprocessor can merge annotations of identical tuples together with one pass over the result of \( Q \).

3. Return \( Q \).

---

**Example 2.4.1** Consider the SWISS-PROT relation of Figure 2.1 and assume that there is an extra attribute \( \text{Size} \). Consider the following pSQL query \( Q \) with custom propagation scheme written against SWISS-PROT:

\[
Q = \text{SELECT } s\text{.ID AS ID, } s\text{.Desc AS Desc, } s\text{.Size AS Size,} \\
\text{FROM SWISS-PROT } s \\
\text{PROPAGATE } s\text{.ID TO Desc, } s\text{.Desc TO Desc, } s\text{.Size TO Size}
\]

Every tuple in SWISS-PROT will be emitted in such a way that the set of annotations associated with the \( \text{Desc} \) column of a tuple in the output is the union of annotations associated with both \( \text{ID} \) and \( \text{Desc} \) of the corresponding tuple in SWISS-PROT. Furthermore, the annotations associated with the \( \text{Size} \) column of a tuple are the same annotations associated with the \( \text{Size} \) column of the corresponding tuple in SWISS-PROT and the column \( \text{ID} \) of every tuple in the output does not carry any annotations.

In Step 1 of algorithm Custom-pSQL-To-SQL, the following two intermediate SQL queries are generated and added to the set since \( \text{bin}(\text{ID}) \) is empty, \( \text{bin}(\text{Desc}) = \{ s\text{.ID}_a \text{ AS Desc}_a, s\text{.Desc}_a \text{ AS Desc}_a \} \) and \( \text{bin}(\text{Size}) = \{ s\text{.Size}_a \text{ AS Size}_a \} \).

\[
Q_1 = \text{SELECT } s\text{.ID AS ID, NULL AS ID}_a s\text{.Desc AS Desc, } s\text{.ID}_a \text{ AS Desc}_a, \\
\text{FROM SWISS-PROT } s \\
Q_2 = \text{SELECT } s\text{.ID AS ID, NULL AS ID}_a s\text{.Desc AS Desc, } s\text{.Desc}_a \text{ AS Desc}_a, \\
\text{FROM SWISS-PROT } s
\]
Observe that the number of SQL queries in $Q$ is always equal to the maximum bin size. In Step 2, the algorithm generates the following wrapper SQL query:

$$Q_s = \text{SELECT DISTINCT } * \text{ FROM } (Q_1 \text{ UNION } Q_2) \text{ ORDER BY } \text{ID, Desc, Size}$$

Observe that $Q_1$ and $Q_2$ are unioned, and the result is sorted according to the attributes in the \textit{selectlist} of $Q$. The tuples are sorted according to the \textit{selectlist} of $Q$ so that the Postprocessor can merge annotations associated with identical cells in the output of $Q$ in one pass over the result of $Q_s$.

2.4.3 The Postprocessor

The Postprocessor scans the set of tuples returned by the RDBMS and unions together the annotations from duplicate tuples for proper display. This operation is done in linear time in the number and size of tuples retrieved, provided that the set of emitted tuples is already sorted. For example, if the postprocessor receives the first table of Section 2.4.1 as input, it returns $\{ (a \{a_1, a_2\}, b \{b_1\}) \}$.

Example 2.4.2 Suppose the following tuples are returned by the database system, sorted according to the attributes $A$ and $B$.

<table>
<thead>
<tr>
<th>A</th>
<th>$A_a$</th>
<th>B</th>
<th>$B_{a_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$a_1$</td>
<td>b</td>
<td>$a_2$</td>
</tr>
<tr>
<td>a</td>
<td>$a_3$</td>
<td>b</td>
<td>$-$</td>
</tr>
<tr>
<td>a</td>
<td>$-$</td>
<td>c</td>
<td>$a_2$</td>
</tr>
</tbody>
</table>

The result returned by the Postprocessor is $\{ (a \{a_1, a_3\}, b \{a_2\}), (a \{\}, c \{a_2\}) \}$.
2.4.4 Other Features of the Prototype

This section summarizes other features implemented in DBNotes, besides its annotation propagation abilities. Figure 2.4 shows a snapshot of DBNotes in a scenario requiring the integration of information about celebrity-owned restaurants from four databases. The visual interface of DBNotes allows one to formulate pSQL queries, and analyze their annotated output. In the annotated output, each row represents a tuple, and the annotations are shown with bullets. In this application, the annotations represent reviews about restaurants.

**Querying Annotations** The pSQL query language of DBNotes has been extended to allow
a user to query the annotations associated with the data, in addition to the data itself. With this feature, a user can pose queries to retrieve, for example, all tuples that are derived from a particular data source, or all tuples with an attached error report. Annotations can be queried through the use of the keyword `ANNOT(attribute-name)`, which elevates annotations of attribute-name to first-class citizens. Some examples of how one could pose queries against data and annotations are shown below.

```
SELECT DISTINCT f.Name AS Celebrity, r.Name AS Restaurant
FROM Frequents f, Blue_Pages r,
    ANNOT(r.Name) a
WHERE f.Restaurant = r.Name AND
    r.Cost = 'Expensive' AND
    a LIKE '%Blue_Pages:Bad%'
PROPAGATE DEFAULT
```

This query retrieves all celebrities that frequent expensive restaurants that received a bad review from Blue_Pages. Here, Blue_Pages and Frequents are two relations containing information about restaurants, and respectively, famous stars and the restaurants they visit. The annotation variable “a” ranges over elements in the set of annotations associated with each Name value of each tuple in Blue_Pages. This query demonstrates how one can query over data and annotations. It also demonstrates how one can query about source information provided that annotations carry information about the sources. Additionally, pSQL can also be used to count the number of annotations. The query below retrieves bad restaurants, where a restaurant is considered as bad if it has more than two bad ratings. Here, Restaurants is a relation which integrates restaurant information from all data sources used in this application.
Visualizing the Provenance and Flow of Data DBNotes has the capability to visually describe the journey (i.e., the provenance and flow) taken by a piece of data through various databases and transformation steps. Figure 2.5 shows snapshots of visual diagrams displayed in DBNotes in the restaurant integration application. When a user asks about the journey of an attribute value $d$ in a database $D$, DBNotes displays all sequences of databases and transformation steps that derive $d$ in $D$ (i.e., the provenance of $d$). For example, the diagram shown in Figure 2.5(a) illustrates that a selected Name value in Restaurants has been copied from Name values in three databases: Restaurant_Reviews, Partial_Restaurants, and Seeing_Stars, via the transformation denoted as $Q$ in the figure. In turn, the value in Partial_Restaurants has been copied from the Blue_Pages and LATimes databases. Moreover, DBNotes can also display all sequences of databases and transformation steps that depend on the selected data $d$ in $D$ (i.e., the flow of
The diagram shown in Figure 2.5(b) illustrates the chains of transformations responsible for copying a selected Name value in Blue_Pages to three other databases.

In computing these diagrams, DBNotes relies on “address” annotations which are automatically maintained by the system as described in Section 2.4.1. DBNotes systematically collects and maintains these annotations as data is being transformed through queries, and determines the sequence of databases and query transformations that derived a value, directly, by analyzing its annotations. The flow of a value, however, cannot be eagerly computed, as we are unaware of what future transformations might depend on the value. Instead, DBNotes maintains a Transformations catalog which records information about source and target databases along with the transformations between them. When computing the flow of a value, the Transformation catalog is leveraged to identify possible transformations which might have subsequently copied the selected value.

2.5 Experimental Evaluation

This section describes experiments that have been conducted to evaluate the feasibility of DBNotes. The main goal of the experimental study was to compare the performance of queries under different propagation schemes (default, default-all, or no propagation scheme (i.e., SQL queries)), as well as compare the performance of queries when the number of annotations in a database is varied. The experiments have been conducted on a Pentium 4, 2.8GHz machine with 1GB RAM.

Our experimental results indicate that on databases annotated up to 100% (i.e., where
each attribute value has at most one annotation), the performance of default queries is comparable to that of SQL queries. The combination of pSQL default queries on databases annotated 100% is an important setting, as it is usually sufficient for tracing provenance purposes. Hence, this result is encouraging, in that it demonstrates the feasibility of DBNotes (and that of the naive storage scheme) in eagerly computing provenance. Our experimental results also indicate that the execution time of pSQL queries with default scheme increases only marginally when the number of annotations in the database is doubled, but still remains below 100%. This offers further evidence that DBNotes’ naive storage scheme is appropriate in scenarios where pSQL with default propagation behavior is desired, and there are at most 100% annotations. However, on databases annotated more than 100% (i.e., where every attribute value has at least one annotation), the performance of pSQL default queries starts to degrade. Our experimental results suggest that the overhead resulting from processing duplicate tuples may not be acceptable in general. Hence, an alternative annotation storage scheme, which does not involve duplication of data, would be more suitable.

Our experimental results also indicate that a pSQL query with default-all propagation scheme may perform significantly slower compared to the same query under the default scheme. This is explained by the fact that each query in the query basis must be executed, and not a single query, in order to collect all relevant annotations under the default-all scheme. Investigating techniques for optimizing the query basis to improve the performance of pSQL queries with default-all propagation behavior remains an interesting direction for future research.
2.5.1 Methodology

Datasets The databases used to perform the experiments are from the TPC Benchmark H (TPCH) Standard Specification Revision 2.1.0 [93]. In our experiments, we used TPCH data of various sizes and we call these databases the unannotated databases. In order to create annotated datasets, we modified the TPCH schema to conform to the naive storage scheme of DBNotes by adding an additional attribute for every attribute of every relation in the TPCH schema. For each unannotated database, three different instances of the modified TPCH database schema corresponding to 30%, 60% and 100% annotated databases have been created. A 30% annotated database means that 30% of the total number of cells in every relation of the database will contain an annotation. The experiments have been conducted on three datasets of sizes 100MB, 500MB and 1GB. Each dataset consists of the unannotated database and the three annotated databases (30%, 60% and 100%). The primary key indices specified in the TPCH standard specification have been created for the unannotated database, as well as the annotated databases. In all our experiments, the default statistics provided by Oracle were available to the query optimizer.

Workload We executed queries of increasing join sizes and with varying number of output attributes to determine how well DBNotes scales for these type of queries. As mentioned in Section 2.2, the number of joins and output attributes of a query are in fact particularly important in our Generate-Query-Basis algorithm. The TPCH queries could not be used for the experiments because they include aggregates and nested queries.

The queries $Q_0, \ldots, Q_4$ which denote queries with zero to four joins, respectively, are
shown in Figure 2.6(a). For example, $Q_2$ denotes the query \( \text{Supplier} \bowtie \text{Nation} \bowtie \text{Region} \) with two joins, on the attributes Nationkey and Regionkey respectively. The cardinality of each relation in the 100MB dataset is shown in brackets. (For the 500MB and 1GB datasets, the cardinalities of relations Nation and Region are the same, while the cardinalities of relations Customer, Supplier and Partsupp are 5 and respectively, 10 times larger.) The workload consists of queries $Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$, which denote the queries with $i$ joins and one, three, and five output attributes, respectively.

**Techniques** The workload queries have been executed under both the default and the default-all schemes on the annotated databases. The SQL query that corresponds to each of these queries has been executed on the unannotated databases, in order to be able to measure the overhead that the propagation of annotations introduces in the overall running time of the queries. All the experiments were performed on a warm buffer and the buffer size was set to 256MB.

We have implemented and experimented with both the optimized, as well as the un-optimized versions of the Generate-Query-Basis algorithm. This section reports only the results obtained with the optimized version, as it consistently and significantly outperformed the unoptimized version.

### 2.5.2 Experimental Results

**Experiment 1** The goal of this experiment is to compare the performance of pSQL queries under different propagation schemes (default, default-all or no propagation scheme). The performance of DBNotes is measured for queries under the default and default-all propagation scheme on the 100% annotated database in each of our three datasets. The workload queries
Figure 2.6: (a) Queries used in our experiments and (b) comparison in performance for default and default-all schemes on the 100MB dataset.

$Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$ have been executed on the 100% annotated databases. The SQL query that corresponds to each of these queries has also been executed on the unannotated databases. The results obtained with the 100MB, 500MB and 1GB datasets are shown in Figures 2.6(b) and 2.7(a–b).

Figure 2.6(b) illustrates the execution time (the total time taken by the translator, RDBMS, and postprocessor to emit all tuples in the result) of each query for the default and default-all propagation scheme on the 100% annotated database in the 100MB dataset. We have observed that as more output attributes are emitted, the execution time of each query under the default scheme (respectively, the default-all scheme) increases only slightly (see, for instance, $Q_0(1), Q_0(3)$, and $Q_0(5)$). The increase in time is due to longer execution time taken by Oracle, as well as additional overhead incurred in postprocessing, as more attributes of different tuples need to be compared. However, for the default-all scheme, the number of SPJ queries that are sent to Oracle increases (2, 4, and 6 SPJ queries, respectively) as the number of output
attributes increases. This has a negative effect on the performance of default-all pSQL queries. Table 2.1 provides the exact execution times of each query for 100% annotated database and the number of SPJ queries that are generated for the default all-scheme. In the worst case, a query such as $Q_4(5)$ may run about 8 times slower than both the query with default scheme and the actual SQL query. This can be explained by the fact that there are 6 SPJ queries, each with four joins, which are generated and sent to Oracle for $Q_4(5)$, instead of a single query.

Figure 2.7: Comparison in performance for default and default-all schemes on the (a) 500MB, and respectively (b) 1GB dataset.
Table 2.1: The execution time of each query for each database in the 100MB dataset and each propagation scheme. The columns “#pSQL” and “#SPJ” denote the size of the query basis and number of SPJ queries that are generated, respectively, for the default-all scheme.

In the best case (e.g., $Q_4(1)$), a query with default-all scheme runs about twice as slow than the same query with default scheme. For the default-all scheme there is no increase in the number of pSQL and SPJ queries that are generated when the number of joins increases, since the attributes that are selected do not participate in the joins. (The performance of default-all pSQL queries where attributes that participate in the joins are selected as well is evaluated in Experiment 3.) The number of pSQL and SPJ queries that are generated increases when the number of output attributes increases and they increase linearly. The execution times of $Q_1(j)$, $j \in [1, 3, 5]$, decreases slightly when compared with $Q_0(j)$ because a join on a small relation has been made.

For the default scheme, the execution times of pSQL queries are comparable to those of SQL queries. On average, the pSQL queries with default scheme used in this experiment took around 40% more time to execute than their corresponding SQL queries, and at best the execution time of a pSQL query with default scheme is the same as the execution time of its
corresponding SQL query (e.g., $Q_4(1)$).

Similar trends have been observed for larger datasets. Figures 2.7(a) and 2.7(b) illustrate the execution time of each query for the default and default-all propagation scheme on the 100% annotated databases in the 500MB and respectively, 1GB datasets. We have observed that the overhead of propagating annotations under the default scheme is smaller on larger datasets. On average, the pSQL queries with default scheme took only 15% and 24% more time to execute than their corresponding SQL queries on the 500MB and respectively, 1GB dataset. This can be explained by the fact that on larger datasets, the postprocessing time tends to become less significant when compared to the time taken by the database engine to execute the queries.

The results of this experiment demonstrate the feasibility of DBNotes in tracing data provenance. Given the benefit offered by annotations in answering provenance, a performance overhead of 40% (and even less on larger databases) for pSQL queries with default scheme seems acceptable in general. However, the performance overhead of default-all queries may not be acceptable in general. In this experiment, the overhead was largely due to the size of the query basis: a default-all pSQL query whose query basis contains $n$ queries executed around $n$ times slower, compared with the same query with default scheme, on databases annotated at most 100%. While this may be acceptable in special cases (e.g., when the running time of the query is small, or the size of the query basis is small), it remains an interesting open question whether the query basis can be optimized to make the default-all scheme feasible in general.

**Experiment 2** This experiment evaluates the influence of the number of annotations in a database on the execution time of pSQL queries under default or default-all schemes. The workload queries $Q_i(1), Q_i(3), Q_i(5), \ i \in [0, 4]$ have been executed under both default and default-
all schemes on the 30\%, 60\% and 100\% annotated databases. The results obtained with the 100MB, 500MB and 1GB datasets are illustrated in Figures 2.8, 2.9 and, respectively, 2.10. (the results obtained with the 100MB dataset are also tabulated in Table 2.1).

Intuitively, we expected an increase in postprocessing time when the number of annotations in the database increases, as more annotations need to be compared and unioned together. Perhaps surprisingly, we have observed only a slight increase in the execution time of
Figure 2.9: Performance comparison for (a) default and (b) default-all pSQL queries on databases annotated in various degrees on the 500MB dataset.

Each query across databases annotated in various degrees. Table 2.2 shows the average percentage increases incurred in the total execution times of the default and default-all queries used in this experiment. Our results indicate that on databases annotated up to 100%, the overhead due to postprocessing of more annotations is small compared to the total execution time of the queries. On the 100MB dataset, for example, the total execution time for default queries increases on average 0.71% when the number of annotations in the database is doubled from 30%
Figure 2.10: Performance comparison for (a) default and (b) default-all pSQL queries on databases annotated in various degrees on the 1GB dataset.

Annotations to 60% annotations and 1.85% when the number of annotations is varied from 60% annotations to 100% annotations. An increase in the number of annotations in the database induces smaller increases in the total execution times of default-all queries when compared to default queries. Since multiple queries are being executed in the default-all case, the postprocessing time becomes less significant when compared to the actual time taken by the engine to execute these queries.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Default</th>
<th></th>
<th>Default-All</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30% to 60% (%) increase</td>
<td>60% to 100% (%) increase</td>
<td>30% to 60% (%) increase</td>
<td>60% to 100% (%) increase</td>
</tr>
<tr>
<td>100MB</td>
<td>0.71%</td>
<td>1.85%</td>
<td>0.10%</td>
<td>0.02%</td>
</tr>
<tr>
<td>500MB</td>
<td>1.26%</td>
<td>1.83%</td>
<td>0.31%</td>
<td>0.65%</td>
</tr>
<tr>
<td>1GB</td>
<td>1.22%</td>
<td>1.36%</td>
<td>0.44%</td>
<td>0.99%</td>
</tr>
</tbody>
</table>

Table 2.2: The average percentage increases incurred in the total execution times of the queries under both default and default-all schemes when the number of annotations in the database is varied from 30% to 60% annotations and from 60% to 100% annotations.

**Experiment 3** This experiment evaluates the effect of selecting attributes that participate in join conditions on the performance of default-all pSQL queries. For this purpose, the execution time of queries $Q_i(1 + j), Q_i(3 + j), Q_i(5 + j), i \in [1, 3], j \in [1, i]$ was measured under the default-all propagation scheme on the annotated databases in the 100MB dataset. These queries are identical to the original workload queries $Q_i(1), Q_i(3), Q_i(5), i \in [1, 3]$, except that their selectlist additionally contains $j$ attributes selected among the attributes that appear in some join condition in the wherelist. For example, consider the query $Q_1(1)$ which computes the join of tables Supplier and Nation on the Nationkey attribute. The query $Q_1(1 + 1)$ is identical to $Q_1(1)$, except that the attribute Nationkey (which does not appear in the selectlist of $Q_1(1)$) appears in the selectlist of $Q_1(1 + 1)$. The execution times of these queries are shown in Figure 2.11 (they are also tabulated in Table 2.3). The execution times of queries $Q_i(1), Q_i(3), Q_i(5), i \in [1, 3]$ are shown as well, for comparison purposes. Table 2.3 also illustrates the number of tuples retrieved by each query before the postprocessing step. The number of output tuples returned by each query after postprocessing is 1,000.

The execution time of the queries under the default-all propagation scheme increases, as more attributes that participate in the joins are selected. On the 30% annotated database...
for example, the query $Q_2(1 + 1)$ runs 3 times slower when compared to $Q_2(1)$ and the query $Q_2(1 + 2)$ takes 15% more time to run compared to $Q_2(1 + 1)$. As the number of selected attributes involved in join conditions increases, more pSQL queries are generated by our Generate-Query-Basis algorithm, hence more SPJ queries are executed. The overall performance overhead is explained only partially by the increase in the number of executed queries, however, as we shall discuss shortly. As shown in Table 2.3, there are 5 and respectively, 7 SPJ queries that are executed in order to retrieve the correct annotations under the default-all scheme for queries $Q_2(1 + 1)$ and $Q_2(1 + 2)$, while only 2 SPJ queries are executed in case of $Q_2(1)$. On average, the queries used in this experiment took about 5.9, 6.2 and 34.5 times more time to
<table>
<thead>
<tr>
<th>Query</th>
<th>Unannotated</th>
<th>30% Def-All</th>
<th>60% Def-All</th>
<th>100% Def-All</th>
<th>%SQL</th>
<th>%SPI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#tuples</td>
<td>exec. time</td>
<td>#tuples</td>
<td>exec. time</td>
<td>#tuples</td>
<td>exec. time</td>
</tr>
<tr>
<td>$Q_1(i)$</td>
<td>1.000</td>
<td>0.025</td>
<td>1.000</td>
<td>0.0658</td>
<td>1.000</td>
<td>0.072</td>
</tr>
<tr>
<td>$Q_1(i + 1)$</td>
<td>1.000</td>
<td>0.125</td>
<td>1.000</td>
<td>0.266</td>
<td>2.000</td>
<td>0.345</td>
</tr>
<tr>
<td>$Q_2(i)$</td>
<td>1.000</td>
<td>0.019</td>
<td>1.000</td>
<td>0.0722</td>
<td>1.000</td>
<td>0.0748</td>
</tr>
<tr>
<td>$Q_2(i + 1)$</td>
<td>1.000</td>
<td>0.062</td>
<td>1.000</td>
<td>0.219</td>
<td>2.000</td>
<td>0.249</td>
</tr>
<tr>
<td>$Q_2(i + 2)$</td>
<td>1.000</td>
<td>0.032</td>
<td>1.000</td>
<td>0.25</td>
<td>8.000</td>
<td>0.688</td>
</tr>
<tr>
<td>$Q_3(i)$</td>
<td>1.000</td>
<td>0.1532</td>
<td>1.000</td>
<td>0.3622</td>
<td>1.000</td>
<td>0.3594</td>
</tr>
<tr>
<td>$Q_3(i + 1)$</td>
<td>1.000</td>
<td>0.188</td>
<td>1.000</td>
<td>8.515</td>
<td>2.000</td>
<td>8.546</td>
</tr>
<tr>
<td>$Q_3(i + 2)$</td>
<td>1.000</td>
<td>0.187</td>
<td>1.000</td>
<td>6.547</td>
<td>8.000</td>
<td>7.359</td>
</tr>
<tr>
<td>$Q_3(i + 3)$</td>
<td>1.000</td>
<td>0.204</td>
<td>22,679</td>
<td>23.516</td>
<td>109,692</td>
<td>37.782</td>
</tr>
<tr>
<td>$Q_1(3)$</td>
<td>1.000</td>
<td>0.0312</td>
<td>1.000</td>
<td>0.0968</td>
<td>1.000</td>
<td>0.0968</td>
</tr>
<tr>
<td>$Q_1(3 + 1)$</td>
<td>1.000</td>
<td>0.063</td>
<td>1.000</td>
<td>0.234</td>
<td>2.000</td>
<td>0.313</td>
</tr>
<tr>
<td>$Q_2(3)$</td>
<td>1.000</td>
<td>0.0284</td>
<td>1.000</td>
<td>0.1002</td>
<td>1.000</td>
<td>0.0998</td>
</tr>
<tr>
<td>$Q_2(3 + 1)$</td>
<td>1.000</td>
<td>0.047</td>
<td>1.000</td>
<td>0.266</td>
<td>2.000</td>
<td>0.328</td>
</tr>
<tr>
<td>$Q_2(3 + 2)$</td>
<td>1.000</td>
<td>0.062</td>
<td>1.000</td>
<td>0.312</td>
<td>8.000</td>
<td>0.796</td>
</tr>
<tr>
<td>$Q_3(3)$</td>
<td>1.000</td>
<td>0.191</td>
<td>1.000</td>
<td>1.1186</td>
<td>1.000</td>
<td>1.1188</td>
</tr>
<tr>
<td>$Q_3(3 + 1)$</td>
<td>1.000</td>
<td>0.187</td>
<td>1.000</td>
<td>6.937</td>
<td>2.000</td>
<td>6.859</td>
</tr>
<tr>
<td>$Q_3(3 + 2)$</td>
<td>1.000</td>
<td>0.219</td>
<td>1.000</td>
<td>8.375</td>
<td>8.000</td>
<td>9.438</td>
</tr>
<tr>
<td>$Q_3(3 + 3)$</td>
<td>1.000</td>
<td>0.219</td>
<td>22,679</td>
<td>30.015</td>
<td>109,692</td>
<td>115.202</td>
</tr>
<tr>
<td>$Q_1(5)$</td>
<td>1.000</td>
<td>0.0438</td>
<td>1.000</td>
<td>0.138</td>
<td>1.000</td>
<td>0.1312</td>
</tr>
<tr>
<td>$Q_1(5 + 1)$</td>
<td>1.000</td>
<td>0.063</td>
<td>1.000</td>
<td>0.313</td>
<td>2.000</td>
<td>0.375</td>
</tr>
<tr>
<td>$Q_2(5)$</td>
<td>1.000</td>
<td>0.0406</td>
<td>1.000</td>
<td>0.1498</td>
<td>1.000</td>
<td>0.1468</td>
</tr>
<tr>
<td>$Q_2(5 + 1)$</td>
<td>1.000</td>
<td>0.063</td>
<td>1.000</td>
<td>0.547</td>
<td>2.000</td>
<td>0.453</td>
</tr>
<tr>
<td>$Q_2(5 + 2)$</td>
<td>1.000</td>
<td>0.063</td>
<td>1.000</td>
<td>0.375</td>
<td>8.000</td>
<td>1.047</td>
</tr>
<tr>
<td>$Q_3(5)$</td>
<td>1.000</td>
<td>0.231</td>
<td>1.000</td>
<td>1.5128</td>
<td>1.000</td>
<td>1.6096</td>
</tr>
<tr>
<td>$Q_3(5 + 1)$</td>
<td>1.000</td>
<td>0.249</td>
<td>1.000</td>
<td>9.047</td>
<td>2.000</td>
<td>9.173</td>
</tr>
<tr>
<td>$Q_3(5 + 2)$</td>
<td>1.000</td>
<td>0.265</td>
<td>1.000</td>
<td>10.953</td>
<td>8.000</td>
<td>12.297</td>
</tr>
<tr>
<td>$Q_3(5 + 3)$</td>
<td>1.000</td>
<td>0.266</td>
<td>22,679</td>
<td>50.563</td>
<td>109,692</td>
<td>197.828</td>
</tr>
</tbody>
</table>

Table 2.3: The execution time of each default-all query from Experiment 3 on the 30%, 60% and 100% annotated databases in the 100MB dataset. The columns “%SQL” and “%SPI” denote the size of the query basis and respectively, the number of SPJ queries that are generated. The columns “#tuples” show the number of tuples retrieved by the queries before the postprocessing phase.

execute on the 30%, 60% and respectively, 100% annotated databases when one join attribute was selected, compared to the same queries with no join attributes appearing in their selectlist.

When two join attributes were selected, the queries ($Q_i(1 + 2), Q_i(3 + 2), Q_i(5 + 2), i \in [2, 3]$) run on average about 1.03, 1.84 and 1.24 times slower on the 30%, 60% and respectively, 100% annotated databases compared to the same queries where only one join attribute was selected (i.e., $Q_i(1 + 1), Q_i(3 + 1), Q_i(5 + 1), i \in [2, 3]$). Finally, the queries which select three join attributes ($Q_3(1 + 3), Q_3(3 + 3), Q_3(5 + 3)$) run about 4, 11 and respectively, 10 times slower on the 30%, 60% and respectively, 100% annotated databases compared to the same queries.
where only two join attributes are selected (i.e., $Q_3(1+2), Q_3(3+2), Q_3(5+2)$).

As the number of selected join attributes increases, not only that there are more SPJ queries that are executed, but the query engine and the postprocessor module are given significantly more tuples to sort and respectively, merge. In the case of query $Q_1(1)$ for example, there are 1,000 tuples that have to be sorted and further postprocessed. However, in the case of query $Q_1(1+1)$ (which additionally selects one join attribute), there are 2,000 and respectively, 41,000 tuples that have to be sorted and postprocessed when this query is run on the 60% and respectively, 100% annotated databases. This explains why $Q_1(1+1)$ runs about 5 and respectively, 38 times slower on the 60% and respectively, 100% databases when compared to $Q_1(1)$. The fact that as many as 41,000 tuples are retrieved (before postprocessing) when query $Q_1(1+1)$ is run on the 100% database was very unexpected. However, our investigations discovered a simple explanation for this fact. Recall that this query performs a join between the tables Supplier and Nation on the Nationkey attribute which is also selected in the output. There are 1,000 tuples in Supplier and 25 distinct values for the attribute Nationkey. Since in the 100% database each value has one distinct annotation, it follows that each distinct Nationkey value in the table Supplier has about 40 distinct annotations. According to our Generate-query-basis algorithm, a query that performs a self join of Supplier on the Nationkey attribute will be executed in order to extract the 40 distinct annotations for each Supplier tuple (these annotations are all needed, according to the semantics of pSQL queries with default-all propagation scheme). This query will clearly generate around 40,000 tuples. Although it seems excessive to pull out all these 40 annotations for each tuple in Supplier, we note however that this situation arose precisely because each Nationkey value had a distinct annotation in the Supplier
Table 2.4: The execution time of each query for each propagation scheme and each database annotated more than 100% in the 100MB dataset. The columns “#pSQL” and “#SPJ” denote the size of the query basis and number of SPJ queries that are generated, respectively, for the default-all scheme.

<table>
<thead>
<tr>
<th>Query</th>
<th>Unannotated</th>
<th>130% Def</th>
<th>130% Def-All</th>
<th>160% Def</th>
<th>160% Def-All</th>
<th>200% Def</th>
<th>200% Def-All</th>
<th>pSQL</th>
<th>SPJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0(1)</td>
<td>0.0282</td>
<td>0.0814</td>
<td>0.1846</td>
<td>0.084</td>
<td>0.1906</td>
<td>0.087</td>
<td>0.1844</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q1(1)</td>
<td>0.025</td>
<td>0.0746</td>
<td>0.122</td>
<td>0.0656</td>
<td>0.1314</td>
<td>0.072</td>
<td>0.1472</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q2(1)</td>
<td>0.019</td>
<td>0.069</td>
<td>0.1376</td>
<td>0.0782</td>
<td>0.1628</td>
<td>0.0908</td>
<td>0.2092</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q3(1)</td>
<td>0.1532</td>
<td>1.933</td>
<td>5.9284</td>
<td>2.875</td>
<td>9.2096</td>
<td>6.8282</td>
<td>37.4134</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q4(1)</td>
<td>92.4604</td>
<td>1644.0896</td>
<td>5029.2506</td>
<td>2648.0438</td>
<td>8392.7502</td>
<td>1682.842</td>
<td>5594.8612</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Q0(3)</td>
<td>0.0252</td>
<td>0.1034</td>
<td>0.4622</td>
<td>0.1062</td>
<td>0.481</td>
<td>0.0966</td>
<td>0.5084</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q1(3)</td>
<td>0.0312</td>
<td>0.103</td>
<td>0.4968</td>
<td>0.1094</td>
<td>0.55</td>
<td>0.1062</td>
<td>0.5534</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q2(3)</td>
<td>0.0284</td>
<td>0.1128</td>
<td>0.5344</td>
<td>0.1126</td>
<td>0.6254</td>
<td>0.1312</td>
<td>0.725</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q3(3)</td>
<td>0.191</td>
<td>2.356</td>
<td>17.9904</td>
<td>3.4972</td>
<td>27.1808</td>
<td>7.569</td>
<td>135.0818</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q4(3)</td>
<td>100.0106</td>
<td>1973.217</td>
<td>16386.8308</td>
<td>3152.887</td>
<td>27529.0176</td>
<td>2324.856</td>
<td>20096.4258</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Q0(5)</td>
<td>0.0502</td>
<td>0.1438</td>
<td>0.953</td>
<td>0.1534</td>
<td>1.0408</td>
<td>1.0594</td>
<td>1.1088</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q1(5)</td>
<td>0.0438</td>
<td>0.1438</td>
<td>1.019</td>
<td>0.147</td>
<td>1.147</td>
<td>0.1498</td>
<td>1.216</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q2(5)</td>
<td>0.0406</td>
<td>0.147</td>
<td>1.1</td>
<td>0.156</td>
<td>1.443</td>
<td>0.175</td>
<td>1.5158</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q3(5)</td>
<td>0.231</td>
<td>2.7968</td>
<td>34.1068</td>
<td>4.153</td>
<td>51.9968</td>
<td>8.2562</td>
<td>346.1802</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q4(5)</td>
<td>111.8918</td>
<td>2347.7426</td>
<td>32228.192</td>
<td>3857.2314</td>
<td>53906.3124</td>
<td>3082.9914</td>
<td>42571.6014</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table. A scenario where we may have one annotation for each Nationkey value is when we are interested in tracing the provenance of data, and each annotation represents an address. In this case, however, the default scheme for propagating annotations is more suitable. Our experimental results show that there may be significant overhead to the default-all scheme when annotations can be excessive.

**Experiment 4** This experiment evaluates the performance of pSQL queries with default and default-all propagation schemes on databases annotated more than 100%. Three additional databases of size 100MB with 130%, 160% and respectively, 200% annotations, have been created for this experiment. In the 130% (respectively, 160%) annotated database, 30% (respectively, 60%) of the values have two annotations, while the rest of the values have only one annotation. In the 200% database, each value has two annotations.

The performance of DBNotes for the workload queries $Q_i(1), Q_i(3), Q_i(5), i \in [0, 4]$ under the default and default-all propagation scheme has been measured on the 130%, 160%
Figure 2.12: Comparison in performance for default and default-all schemes on the 100MB database annotated 200% and 200% annotated 100MB databases. The results obtained are tabulated in Table 2.4. For comparison purposes, Table 2.4 also shows the execution time of the corresponding SQL queries on the unannotated 100MB database.

Figure 2.12 illustrates the execution time of each query for the default and default-all propagation scheme on the 100MB database with 200% annotations. The execution time of each query under the default scheme (and respectively, default-all scheme) increases as more output attributes are emitted. As previously explained (Experiment 1), this is due to longer execution time taken by Oracle to pull out the columns storing the annotations, additional overhead due to
Figure 2.13: Performance comparison for default (a) and default-all (b) pSQL queries on 100MB databases with more than 100% annotations.

postprocessing of annotations, and an increase in the number of SPJ queries that are generated and executed (for the default-all scheme). However, the overhead measured for default queries (compared to their corresponding SQL queries) was much more drastic on the 200% annotated database, compared to the overhead of the same queries on the databases annotated up to 100%.

On average, a pSQL default query $Q_i(1), Q_i(3), Q_i(5), i \in [0, 1, 2, 4]$ took between 3 times (e.g., queries with 0 or 1 joins) and 25 times (e.g., queries with 4 joins) more time to execute compared to their corresponding SQL queries. This is a negative consequence of the naive storage scheme of DBNotes, since each 200% annotated relation has double the number of tuples compared to the same relation with no annotations. This leads to longer postprocessing time as well as longer execution time taken by the query engine, as double the number of tuples have to be processed from each relation. Also note that the more joins in the query, the longer the execution time taken by the query engine. Hence, while the naive scheme has reasonable performance for pSQL queries with default scheme on databases annotated up to 100%, this is no longer the case for databases annotated more than 100%. In such cases, an annotation
storage scheme which does not involve the duplication of data would be more suitable.

Under the default-all scheme, a query such as $Q_4(5)$ may run around 13 times slower when compared to the same query with default propagation scheme, in the worst case. In the best case (e.g., $Q_1(1)$) a query with default-all scheme runs about twice as slow then the same query under the default scheme. Under both default and default-all schemes, the queries with 3 joins (i.e., $Q_3(1)$, $Q_3(3)$ and $Q_3(5)$) behaved unexpectedly. Under the default scheme, these queries ran about 39 times slower (on average) compared to their corresponding SQL queries. The queries $Q_3(3)$ and $Q_3(5)$ took about 17 and respectively, 42 times longer to execute under the default-all scheme when compared to the default scheme. While investigating this issue we discovered that the anomaly arises because Oracle chose poor execution plans for these particular queries.

Figure 2.13 shows the execution times of the queries with default and default-all schemes on the 100MB databases with 100%, 130%, 160% and 200% annotations. On average, the queries with default scheme run 6 times slower when the number of annotations was increased from 100% to 130%. This is due to two factors. First, there are 30% more tuples in the 130% annotated database compared to the 100% annotated database. Second, Oracle chose a poorer plan for executing the queries on the 130% database, with a different join ordering as well as different join algorithms. The plan built for the 130% annotated database involved the nested loops algorithm, while hash joins were used in the plan constructed for the 100% annotated database. By tweaking the Oracle optimizer, we were able to detect that the plan built for the 100% annotated database (using hash joins only) performed much better on the 130% annotated database compared to the plan chosen by the optimizer (which involved nested loops).
The queries with default scheme run on average about 1.2 times slower when the number of annotations was increased from 130% to 160% annotations, as well as from 160% to 200% annotations. This increase in the execution time is mostly due to the fact that the number of tuples in the database increases with the number of annotations. In general, the plan chosen by the optimizer for the 160% to 200% annotated databases was the same as the plan chosen for the 130% annotated database. For the queries with 3 joins (i.e. $Q_3(1), Q_3(3)$ and $Q_3(5)$), the optimizer chose a poorer plan on the 160% annotated database when compared to the plans generated for both the 130% and 200% annotated databases. This explains why these queries (under both default and default-all propagation schemes) run slower on the 160% annotated database than on both the 130% and 200% annotated databases. On average, the queries with default-all scheme took about 13, 1.3 and respectively 1.8 times longer to execute when the number of annotations was increased from 100% to 130% annotations, 130% to 160% annotations, and respectively, from 160% to 200% annotations.

### 2.6 Discussion

#### 2.6.1 Other Possible Schemes for Managing Annotations

Besides our naive storage scheme, there are other possible schemes for storing and managing annotations. We briefly discuss two of them next.

**Annotation-Relation Storage Scheme** In this scheme, annotations of a relation $R$ are stored in a separate relation $RA$, which we call the ‘annotation-relation of $R$’. The basic schema of $RA$ has three attributes (id, attribute, annotation) where an id value uniquely identifies a tuple in $R$. 

90
a name value is an attribute name in the schema of \( R \) and an annotation value is an annotation of the cell (id, name). An id can either be the primary key of relation \( R \), in which case \( RA \) may have more than three attributes, or some unique identifier used in the database system (e.g., \textit{rowid} in Oracle). For example, to store the tuples \( \{ (a \{a_1, a_2\}, b, \{b_1\}), (c, d) \} \) of the relation \( R(A, B) \) with \( A \) as the key of the relation, we would have an annotation-relation \( RA(id, attribute, annotation) \) with the following tuples: \( (a, A, a_1), (a, A, a_2), (a, B, b_1) \). Trade-offs between the naive and the annotation-relation storage schemes have been investigated in [5].

**Nested Sets Approach**  The multiplicity of a tuple in the naive storage scheme depends on the number of annotations associated with that tuple. A natural approach would be to store annotations associated with each cell as nested sets (i.e., the relation \( R(A, B) \) would be stored as \( R'(A, A_a, B, B_a) \), where \( A_a \) and \( B_a \) are of type nested set). At the time DBNotes was developed, the nested set approach was not yet feasible. None of the commercial databases with support for nested sets offered satisfactory implementations of the operations needed in translating pSQL queries against an underlying storage scheme based on nested sets. As an example, the annotation union operation (i.e., the operation of merging duplicate tuples and their corresponding annotations together) was not direct and had to be performed in several steps in Oracle 10g.

### 2.6.2 Extensions

The language of pSQL queries considered in this chapter does not allow aggregates and bag semantics (i.e., the \textit{DISTINCT} keyword must be present). This section discusses briefly how pSQL can be extended to handle aggregates and bag queries as well.
Aggregates For the default propagation scheme, a possible semantics one may assume for pSQL queries with aggregates is as follows. For aggregates such as count, sum, and average, no annotations are associated with the result of these aggregates, since these aggregate values are not copied from any source values. However, for aggregates such as min\((a)\) and max\((a)\), where \(a\) is an attribute name, the annotations associated with the cell of the resulting min (or max) value are the union of all annotations of the corresponding \(a\)-values whose value equals to the min (or max) value. Alternatively, one may assume the semantics that an output aggregate value is associated with the union of all annotations of all input values that “contribute” to an output aggregate value. Under this alternative semantics, for all aggregate functions count\((a)\), sum\((a)\), average\((a)\), min\((a)\), max\((a)\), the annotations associated with the cell of the resulting value are the union of all annotations of the corresponding \(a\)-values in the input database. An interesting open question is whether the default-all propagation scheme for pSQL queries with aggregates can be achieved.

Bag Semantics It is known from [29] that two conjunctive queries are equivalent under bag semantics if and only if they are isomorphic. This result of [29] implies that to propagate annotations for a pSQL query under the default-all propagation scheme and bag semantics, it suffices to generate only the representative query of that pSQL query in Algorithm Generate-Query-Basis. To handle bag queries, however, the naive storage scheme can no longer be used since the multiplicity of a tuple in this storage scheme depends on the number of annotations that are associated with that tuple. An alternative storage scheme that does not modify the original relation is needed (e.g., store every annotation and its cell in a separate relation). To propagate annotations under the default-all propagation scheme and bag semantics for unions
of conjunctive queries, however, one must first provide a characterization of bag equivalence for unions of conjunctive queries.

2.7 Related Work

The problem of computing data provenance is not new. Figure 2.14 illustrates a classification of systems for tracing the provenance of data over SQL queries. (These systems are also listed in Figure 1.3(c).) Cui et al. [38] first approached the problem of tracing the provenance of data that is the result of an SQL query applied on a relational database. The solution proposed in [38] was to first generate a “reverse” query $Q^r$ when asked to compute the provenance of an output tuple $t$ in the result of a query $Q$ applied on a database $D$ (i.e., $Q(D)$). The result of applying $Q^r$ on $D$ consists of all combinations of source tuples in $D$ such that each combination of source tuples and $Q$ explains why $t$ is in the output of $Q(D)$. The type of provenance studied by [38] is called why-provenance according to Buneman et al. [27]. Additionally, it is desirable to know where the values of a tuple $t$ in the result of $Q(D)$ are copied from in $D$. The latter type of provenance is called where-provenance in [27] and it is this type of provenance that is used in DBNotes for determining where annotations are propagated from. In both works [27, 38], a
“reverse” query is generated in order to answer provenance. While the reverse query approach works well in general, it requires a reverse query to be generated and evaluated on the source database, every time the provenance of an output tuple is sought for. Hence, the method cannot be applied if the source database becomes unavailable.

The reverse query approach is a lazy approach for computing provenance; a query is generated and executed to compute the provenance only when needed. In contrast, DBNotes uses the eager approach for tracing provenance: the provenance of data is carried along (in the form of annotations) as data is being transformed. Hence, the provenance of data is eagerly computed and immediately available in the output. The idea of eagerly computing provenance by forwarding annotations along data transformations is also not new and has been proposed in various forms in existing literature [18, 64, 98]. In fact, the default annotation propagation rules of DBNotes, which propagate annotations based on where-provenance, are similar to those implemented in the Polygen system [98]. In Polygen, however, only information about which source relations a value is copied from is propagated along. In contrast, DBNotes is flexible in the amount of information that is carried along to the result (i.e., it could be the source relations, or the exact cell within the source relations, or a comment on the data). Furthermore, DBNotes provides additional annotation propagation schemes (i.e., the custom, and default-all schemes), as well as other features such as querying annotations (Section 2.4.4).

In DBNotes, an annotation may be attached to a single attribute value in a tuple. The MONDRIAN system [49, 50] extends DBNotes by allowing annotations to be placed on sets of attribute values in a tuple. Trio [4, 16, 83] is a system for managing provenance and uncertainty in relational databases. In Trio, a special form of annotation is automatically attached to each
tuple in a relation, to describe the “address” of that tuple in the database. These annotations
are automatically propagated to the output, as data is being transformed through a query. The
way annotations propagate in Trio is essentially based on why-provenance. The aggregate of
annotations associated with an output tuple is used to determine the degree of uncertainty of
that output tuple.

Numerous annotation systems have been built to support and manage annotations on
text and HTML documents [59, 63, 74, 84]. Recently, annotation systems for genomic se-
quences [24, 42, 61] have also been built. Laliberte and Braverman [63] discussed how to use
the HTTP protocol to design a scalable annotation system for HTML pages. Schickler et al. [84]
discussed the use of a specialized proxy module that would merge annotations from an annota-
tion store onto a Web page that is being retrieved before sending it to the client browser. An-
notea [59] is a W3C effort to support annotations on any Web document. Annotations are also
stored on annotation servers and XPointer is used for pinpointing locations on a Web document.
A specialized client browser that can understand, communicate, and merge annotations residing
in the annotation servers with Web documents is used. Phelps and Wilensky [73, 74, 75] also
discussed the use of annotations with certain desirable properties in the Multivalent Document
System [75], which supports documents of different media types, such as images, postscript, or
HTML. DAS or BioDAS [24, 42] and the Human Genome Browser [61] are specialized anno-
tation systems for genomic sequence data. In almost all of these systems, the design includes
multiple distributed annotation servers for storing annotations and data is merged from various
sources to display it graphically to an end user. The research of these systems has been focused
on the scalability of design, distributed support for annotations, or other added features.
The idea of an annotation management system in which annotations can be made on relational data was first proposed in [28, 91]. Unlike Web pages, the rigid structure of relations makes it easy to describe the exact position where an annotation is attached. Web pages, however, are often retrieved in part or as a whole. Hence, the issue of what annotations to propagate along when a web page is retrieved is straightforward. In contrast, an annotated relation may undergo a complex transformation as a result of executing a query. Thus, DBNotes is concerned with how annotations should propagate when such complex transformations occur. To the best of our knowledge, DBNotes is the first implementation of an annotation management system for relational databases that allows a user to specify how annotations should propagate, and provides an annotation propagation scheme that is invariant under equivalent query rewriting.
Chapter 3

The SPIDER System

Chapter 2 describes a principled method for eagerly tracing data provenance over SQL queries that we have implemented in the DBNotes system. DBNotes operates in special (and common) data interoperability scenarios where the databases involved are relational, and the mappings between these databases are expressed using SQL queries. In this chapter, we describe a principled method for lazily tracing data provenance over schema mappings [32]. We also describe SPIDER [8], a Schema mapping DEbugger system that we have implemented based on this method. In contrast to DBNotes, SPIDER works in more general data interoperability scenarios where databases can be relational or hierarchical, and the relationships between these databases are expressed as schema mappings, as opposed to SQL queries. Furthermore, the two systems differ fundamentally in the method by which they compute provenance: SPIDER adopts a lazy approach, whereas DBNotes adopts the eager approach.

In computing provenance, both lazy and eager approaches are useful in different scenarios, as discussed in Section 1.2. By adopting a lazy approach, an advantage of SPIDER
when compared to DBNotes, or other eager approaches for that matter, is that SPIDER does not require the reengineering of the mapping for the purpose of carrying extra information to the output database. However, a disadvantage of SPIDER, and lazy approaches in general, is that the process of deriving provenance requires the examination of the source instance. Hence, SPIDER cannot be applied in scenarios when the source data becomes unavailable. In SPIDER, our motivation for studying the lazy approach for computing provenance over schema mappings was two-fold. First, we wanted to develop a facility for understanding and debugging schema mappings that can be readily deployed on existing data exchange and data integration systems based on schema mappings (e.g., Clio [56], or RDA [81]). Although schema mappings based on the formalism of tgds and egds have been used in specifying relationships between schemas in such systems, developmental support for programming schema mappings in this language was still lacking prior to the development of SPIDER. Furthermore, only the eager approach for computing provenance had been studied [96]. To date, SPIDER is the only system for understanding and debugging schema mappings that adopts a lazy approach for computing provenance over schema mappings.

SPIDER facilitates a user’s understanding of the quality of data and schema mappings used in database interoperability scenarios through the display of *routes*. Routes are a form of provenance: they describe the relationships between source and target data with the schema mapping. A user is able to select a set of tuples (or facts) in the target and SPIDER is able to compute routes for the selected target tuples. This chapter describes two algorithms for computing routes that we have developed and subsequently implemented in SPIDER. The input to both algorithms consists of the source instance, the target instance and the selected target tuples.
The schema mapping is fixed.) The first algorithm constructs a concise representation of all routes for the selected target tuples. A main result in this chapter shows that this representation is complete, in the sense that every route for the selected tuples is essentially “embedded” in this representation. Furthermore, the representation has polynomial size, and it is obtained in polynomial time, even when there are exponentially many routes. The second algorithm computes one route fast for the selected target data if there is one, and produces another route if necessary. Another main result in this chapter shows that this algorithm is also complete, and executes in polynomial time in the size of the input.

A prototype implementation of SPIDER has been deployed on top of the Clio system, with Clio’s schema mappings language, which is based on tgds and egds. This chapter describes an experimental study conducted to evaluate the performance of SPIDER’s algorithms for computing routes. Our experimental results show that computing one route performs well, and can be achieved at interactive speeds, in general. Furthermore, computing all routes can also be achieved at interactive speeds when the number of selected target tuples is small, or the source and target instances used are reasonably small. In scenarios where understanding and debugging mappings is desired, this is a realistic assumption, since one would expect a designer to use (small) test instances, and not request the computation of routes for a large number of tuples at a time in such scenarios.

The next section describes the formalism of schema mappings based on tgds and egds. It also illustrates how routes, the main feature of SPIDER, facilitate the process of understanding and debugging schema mappings.
3.1 Background

Schema Mapping A schema mapping is a triple \( \mathcal{M} = (S, T, \Sigma_{st} \cup \Sigma_t) \) [62], where \( S \) is a source schema, \( T \) is a target schema, \( \Sigma_{st} \) is a set of source-to-target dependencies (s-t dependencies) and \( \Sigma_t \) is a set of target dependencies. In the relational-to-relational data exchange framework [44], s-t dependencies is a finite set of s-t tuple generating dependencies (tgds) and the set of target dependencies is the union of a finite set of target tgds with a finite set of target equality generating dependencies (egds). A s-t tgd has the form \( \forall x. \phi(x) \rightarrow \exists y. \psi(x, y) \), where \( \phi(x) \) is a conjunction of atomic formulas over \( S \) and \( \psi(x, y) \) is a conjunction of atomic formulas over \( T \). A target tgd has a similar form, except that \( \phi(x) \) is a conjunction of atomic formulas over \( T \). A target egd is of the form \( \forall x. \phi_T(x) \rightarrow x_1 = x_2 \), where \( \phi_T(x) \) is a conjunction of atomic formulas over \( T \), and \( x_1 \) and \( x_2 \) are variables that occur in \( x \). In data integration [65], schema mappings between the source and integrated schemas are as above, except that the set \( \Sigma_t \) is usually empty (i.e., there are no constraints on the integrated schema).

Figure 3.1 shows a schema mapping \( \mathcal{M} \), where \( S \) consists of the relation schemas Cards, SupplementaryCards, FBAccounts, and CreditCards in Manhattan Credit and Fargo Bank. The target schema consists of the relation schemas Accounts and Clients in Fargo Finance. \( \Sigma_{st} \) consists of three s-t tgds, illustrated as \( m_1, m_2 \) and \( m_3 \) (shown in the box). Only \( m_3 \) is not depicted as arrows in the figure. \( \Sigma_t \) consists of two target tgds \( m_4 \) and \( m_5 \), and a target egd \( m_6 \) (also shown in the box). For conciseness, the universal quantifiers of the dependencies have been omitted in the figure. In this example scenario, the goal is to migrate every cardholder and supplementary card holder of Manhattan Credit as a client of Fargo Finance. Also,
every credit card holder of Fargo Bank is a client of Fargo Finance. Intuitively, \( m_1 \) and \( m_2 \) migrate data from Manhattan Credit to Fargo Finance, while \( m_3 \) migrates data from Fargo Bank to Fargo Finance. For example, \( m_1 \) projects every tuple (or fact) from the Cards relation into two tuples, in Accounts and Clients relations respectively. The target tgds \( m_4 \) and \( m_5 \) state that an account with accHolder value \( s \) exists in Accounts if and only if a client with ssn value \( s \) exists in Clients. Furthermore, the target egd \( m_6 \) states that there can only be one credit limit for an account holder. Although not part of the schema mapping, there is a constraint, depicted as \( f_1 \) in Figure 3.1, that states that for every supplementary card, there must be a sponsoring
Figure 3.2: A source instance $I$ and a solution $J$ for $I$ with the schema mapping from Figure 3.1. 

Card in the **Cards** relation whose **cardNo** equals **accNo**. There is also a constraint $f_2$ from **CreditCards** to **FBAccounts** that states that every credit card holder of Fargo Bank must have a bank account.

**Solutions and homomorphisms** Figure 3.2 illustrates a source instance $I$, as well as a solution $J$ for $I$ under $M$ of Figure 3.1. An instance $J$ is a solution for $I$ under $M$ if $J$ is a finite target instance such that $(I, J)$ satisfies $\Sigma_{st} \cup \Sigma_t$. In other words, $(I, J)$ satisfies the schema mapping $M$. The solution $J$ may contain labeled nulls. In Figure 3.2, $N_1, M_1, ..., M_5, I_1,$ and $A_1$ are labeled nulls. Distinct labeled nulls are used to denote possibly different unknown values in the target instance.

Let $K$ and $K'$ be two instances. A homomorphism $h$ from $K$ to $K'$, denoted as $h: K \rightarrow K'$, is an assignment of constants and labeled nulls of $K$ to the constants and labeled
nulls of $K'$ such that $h(c) = c$ for every constant $c$, and for every tuple (or fact) $R(t)$ of $K$, we have that $R(h(t))$ is a tuple of $K'$. A homomorphism $h$ from a formula $\phi(x)$ to an instance $K$, denoted as $h : \phi(x) \rightarrow K$, is an assignment which maps the variables of $x$ to constants or labeled nulls in $K$ such that for every relational atom $R(y)$ that occurs in $\phi(x)$, $R(h(y))$ is a tuple in $K$.

In general, there are many possible solutions for $I$ under $M$. A universal solution $J$ for $I$ under $M$ has the property that it is a solution, and furthermore it is the most general solution in that there is a homomorphism from $J$ to every solution for $I$ under $M$. It was shown in [44] that the result of chasing $I$ with $\Sigma_{st} \cup \Sigma_t$ is a universal solution.

The schema mapping language in data exchange and integration systems such as Clio [56] and RDA [81] is a nested relational extension of tgds and egds, which also handles the hierarchical model, in addition to the relational model. In these systems, a user makes associations between source and target schema elements by specifying correspondences between these elements. Possible correspondences may also be suggested by the system, based on schema matching techniques [80]. These correspondences, which are illustrated as arrows as in Figure 3.1, are interpreted into s-t (nested) tgds. From these s-t tgds, executable queries (e.g., SQL, or XQuery) are generated. Given a source instance $I$, a solution $J$ is created by applying the generated query on $I$. We note that the current implementations of Clio and RDA do not handle target egds, although the general framework does not impose this restriction.
3.1.1 Example Debugging Scenarios

This section illustrates a few usage scenarios with SPIDER. Suppose Alice, a banking specialist, is interested in understanding and debugging the schema mapping $M$ of Figure 3.1. In most cases, Alice is expected to debug $M$ by providing her own (small) test data for the source. In this case, she uses the source instance $I$ and solution $J$ shown in Figure 3.1.

**Scenario 1: Incomplete and incorrect associations between source and target schema elements**

When Alice browses through $J$, she discovers that the address value of the tuple $t_5$ in Clients contains a null $A$. Knowing that neither Fargo Bank nor Manhattan Credit would allow any of its customers to open an account without providing an address, she probes $t_5$. SPIDER shows a route from the source that is a witness for $t_5$ in the target, depicted as $s_1 \xrightarrow{m_1,h} t_1, t_5$.

The route consists of the source tuple $s_1$ in Cards, the tgd $m_1$, as well as an assignment $h$ of variables of $m_1$: \{ $cn \mapsto 6689$, $l \mapsto 15K$, $s \mapsto 434$, $n \mapsto J$. Long, $m \mapsto Smith$, $sal \mapsto 50K$, $loc \mapsto Seattle$, $A \mapsto A_1$ \}. Under this assignment, the right-hand-side (RHS) of $m_1$ is $t_1$ and $t_5$. Hence, $m_1$ asserts the presence of $t_5$ with $s_1$ and $h$. With this route, Alice discovers that the address of J. Long (i.e., the value “Seattle”) was not copied over by the tgd. Indeed, Figure 3.1 shows that there is no correspondence between any schema element of Cards and address of Clients. Suppose Alice also noticed that in $t_5$, the name value is the same as its maidenName value (i.e., Smith). With the same route $s_1 \xrightarrow{m_1,h} t_1, t_5$, Alice discovers that maidenName of Cards has been incorrectly mapped to name of Clients. She therefore corrects $m_1$ to the following tgd $m'_1$ which (1) adds the missing value correspondence between location of Cards and address of Clients and (2) retrieves the name of Clients from the name of Cards:
$m'_1$: Cards($cn, l, s, n, m, sal, loc$) $\rightarrow$ Accounts($cn, l, s$) $\land$ Clients($s, [n, m, sal, loc]$)

In this scenario, SPIDER has helped Alice discover an incomplete, as well as an incorrect association between source and target schema elements. Ideally, it would be useful to simultaneously demonstrate how the modification of $m_1$ to $m'_1$ affects the tuples in $J$. This is an interesting direction for future research.

**Scenario 2: Incomplete associations between source schema elements** When browsing through $J$, Alice discovered that A. Long (tuple $t_7$) who has an income of 30K, has a credit limit of 40K (tuple $t_4$). Knowing that it is very unlikely for an account holder to have a credit limit that is higher than her income, Alice probes $t_4$. SPIDER explains that $t_4$ was created due to the $s_4$ tuple in FBAccounts and the $s_6$ tuple of CreditCards through $m_3$. Suppose Alice could not find anything peculiar with this explanation. She now requests to view all routes for $t_4$. SPIDER reports only one other route that uses the first tuple in FBAccounts ($s_3$) and the tuple $s_6$ through $m_3$. Since the ssn values of these two source tuples are different, Alice realizes that $m_3$ has missed the join condition on ssn in the source relations. She corrects the s-t tgd to:

$m'_3$: FBAccounts($bn, [cs] n, i, a$) $\land$ CreditCards($cn, cl, cs$)

$$\rightarrow \exists M(\text{Accounts}(cn, cl, cs) \land \text{Clients}(cs, n, M, i, a))$$

Alice may also decide to enforce ssn as a key of the relation Clients, which can be expressed as an egd. In this case, SPIDER has helped Alice discover a missing join condition, as well as realize an additional dependency that may need to be added to the target.

**Scenario 3: Incomplete associations between relations** As Alice browses through the target instance further, she sees that the accNo of the account holder 234 is unspecified ($N_1$ of tuple
As it is not likely that there is no account number for an account holder, Alice probes $N_1$ of $t_2$. SPIDER shows that $t_2$ was created through the target tgd $m_5$ with the tuple $t_6$. With this explanation, the existentially-quantified variable $L$ of $m_5$ is assumed to map to the value $2K$ of $t_2$. Furthermore, SPIDER shows that $t_6$ was created through the tgd $m_2$ with the source tuple $s_2$.

With this information, Alice discovers that $m_2$ is in fact missing an association with the source relation Cards. Indeed, every supplementary card holder must have a sponsoring card holder in Cards and they share the same credit limit. So Alice corrects $m_2$ by adding the association between SupplementaryCards and Cards, as indicated by the constraint $f_1$. Furthermore, the target now includes an Accounts relation that is used to hold the account number and ssn of the supplementary card holder, as well as the credit limit of the sponsoring card holder. The new tgd $m_2'$, with the changes highlighted, is now specified as follows:

$$m_2': \text{Cards}(cn, l, s_1, n_1, m, sal, loc) \land \text{SupplementaryCards}(cn, s_2, n_2, a)$$

$$\rightarrow \exists M \exists I (\text{Clients}(s_2, n_2, M, I, a) \land \text{Accounts}(cn, l, s_2))$$

In this scenario, SPIDER has helped Alice discover an incomplete tgd that misses out on some associations between relations in the source schema, as well as between relations in the target schema. Alice may also choose to remove the tgd $m_2$ completely if she thinks it is incorrect, i.e., only primary card holders of Manhattan Credit are automatically customers of Fargo Finance.

Several remarks are in order now. First, debugging a schema mapping is not solely a matter of identifying the target schema elements that are left unmapped or that are mapped incorrectly from the source. Indeed, as Scenarios 2 and 3 illustrate, the problems may also be
due to missing associations between source schema elements, or missing relations. Second, routes are always computed in their entirety, even though only part of a route may demonstrate problems with the schema mapping, as illustrated in Scenario 3. Third, as illustrated in Scenario 2, observe that there are situations in which a computed route may not reveal any problems with the schema mapping. In Scenario 2, Alice needs the knowledge of the second route for \( t_7 \) to discover the problem in the s-t tgd \( m_3 \). Certainly, one may argue that if the second route for \( t_7 \) would have been computed before the first one, Alice would have been able to debug \( m_3 \) without the knowledge of additional routes for \( t_7 \). It is conceivable, however, that \( t_7 \) is a tuple containing sensitive information and in this situation, the knowledge of all routes for \( t_7 \) would be crucial for the purpose of identifying tgd that export sensitive information. It is also worth mentioning that the above debugging scenarios only illustrate the use of routes for understanding the schema mapping through anomalous tuples. However, routes for correct tuples are also useful for understanding the schema mapping in general.

The next section describes the algorithms behind computing all routes or one route for selected target data for relational-to-relational schema mappings. An extension of these algorithms to handle relational/XML-to-relational/XML schema mappings is discussed in Section 3.3. Another extension of these algorithms to compute routes starting from selected source data is briefly described in Section 3.4.1.
3.2 Route Algorithms

This section formalizes the notion of a route and describes algorithms, as well as the properties of these algorithms for computing all routes or one route for a selected set of tuples. A route illustrates the relationship between source and target data with the schema mapping. As an example, consider again the route for $t_2$ described in Scenario 3 of Section 3.1.1. The route for $t_2$ shows the relationship $s_2 \xrightarrow{m_2} t_6 \xrightarrow{m_5} t_2$. The route shows that $s_2$ and $t_6$ satisfy the tgd $m_2$, and $t_6$ and $t_2$ satisfy the tgd $m_5$. More specifically, a route is a sequence of satisfaction steps, which are defined next.

**Definition 3.2.1 (Satisfaction step)** Let $\sigma$ be a tgd $\forall x \phi(x) \rightarrow \exists y \psi(x,y)$. Let $K$ and $K_1$ be instances such that $K$ contains $K_1$ and $K$ satisfies $\sigma$. Let $h$ be a homomorphism from $\phi(x) \land \psi(x,y)$ to $K$ such that $h$ is also a homomorphism from $\phi(x)$ to $K_1$. We say that $\sigma$ can be satisfied on $K_1$ with homomorphism $h$ and solution $K$, or simply $\sigma$ can be satisfied on $K_1$ with homomorphism $h$, if $K$ is understood from the context. The result of satisfying $\sigma$ on $K_1$ with homomorphism $h$ is $K_2$, where $K_2 = K_1 \cup h(\psi(x,y))$ and $h(\psi(x,y)) = \{ R(h(z)) | R(z) \}$ is a relation atom in $\psi(x,y)$. We denote this step as $K_1 \xrightarrow{\sigma,h} K_2$.

**Example 3.2.1** In the example described earlier with $s_2 \xrightarrow{m_2} t_6 \xrightarrow{m_5} t_2$, the first satisfaction step is $({s_2}, \emptyset) \xrightarrow{m_2,h_1} ({s_2}, \{t_6\})$, where $h_1 = \{ \text{an} \mapsto 6689, \text{s} \mapsto 234, \text{n} \mapsto \text{A.Long}, \text{a} \mapsto \text{California}, \text{M} \mapsto M_1, \text{I} \mapsto I_1 \}$. The result of satisfying $m_2$ on the instance $({s_2}, \emptyset)$ with homomorphism $h_1$ and solution $J$ of Figure 3.2 is $({s_2}, \{t_6\})$.

In the context of routes, the instances $K$ and $K_1$ in Definition 3.2.1 are instances over the schema $(S, T)$, not necessarily satisfying the source or target constraints. A few technical
differences between a satisfaction step, a chase step [44], and a solution-aware chase step [48] are described next. First, unlike the definition of a chase step or solution-aware chase step for a tgd \( \forall x \phi(x) \rightarrow \psi(x, y) \) where \( h \) is defined only for \( x \), the homomorphism \( h \) in a satisfaction step is defined for variables in both \( x \) and \( y \). In other words, \( \phi(x) \land \psi(x, y) \) is completely defined under \( h \). Second, when \( \sigma \) is satisfied on \( K_1 \) with homomorphism \( h \), it may be that \( \sigma \) is already satisfied on \( K_1 \) with some other homomorphism \( h_0 \) where \( h(x) = h_0(x) \) for every \( x \in x \). This is allowed in order to be able to use \( \sigma \) again, with a different homomorphism, to witness the existence of some other tuple not in \( K_1 \). Third, there is no corresponding definition for egds. This is because if \( K \) already satisfies an egd \( \sigma \), then \( K_1 \) must also satisfy \( \sigma \) since it is contained in \( K \).

**Definition 3.2.2 (Route)** Let \( M = (S, T, \Sigma_s, \Sigma_t) \) be a schema mapping, \( I \) be a source instance and \( J \) be a solution of \( I \) under \( M \). Let \( J_s \subseteq J \). A route for \( J_s \) with \( M \), \( I \) and \( J \) (in short, a route for \( J_s \)) is a finite non-empty sequence of satisfaction steps \( (I, \emptyset) \xrightarrow{m_1, h_1} (I, J_1) \ldots (I, J_{n-1}) \xrightarrow{m_{n-1}, h_{n-1}} (I, J_n) \), where (a) \( J_i \subseteq J, 1 \leq i \leq n \), (b) \( m_i, 1 \leq i \leq n - 1 \), are among \( \Sigma_s \cup \Sigma_t \), and (c) \( J_s \subseteq J_n \). We say that the set of tuples produced by this route is \( J_n \).

**Example 3.2.2** Referring to Example 3.2.1 and the source instance \( I \) and solution \( J \) of Figure 3.2, \( (I, \emptyset) \xrightarrow{m_2, h_1} (I, \{t_6\}) \) is a route for \( t_6 \). The following is also a route for \( t_6 \): \( (I, \emptyset) \xrightarrow{m_2, h_1} (I, \{t_6\}) \xrightarrow{m_5, h_2} (I, \{t_6, t_2\}) \), where \( h_2 = \{s \mapsto 234, n \mapsto A.Long, m \mapsto M_1, i \mapsto I_1, a \mapsto California, N \mapsto N_1, L \mapsto 2K\} \). The target instance produced by the second route for \( t_6 \) is \( \{t_2, t_6\} \) and the last satisfaction step is redundant, in the sense that the first satisfaction step is sufficient as a route for \( t_6 \).
Observe that our definition of a satisfaction step (and hence a route) is not tied to the chase procedure that has been defined for data exchange [44], or the execution engine of systems such as Clio, or RDA for that matter. The definition is based on the logical satisfaction of tgd. To illustrate this, consider the following example:

**Example 3.2.3** Consider the following schema mapping $\mathcal{M}$.

$$\Sigma_{st}: \quad \text{EmpPhone}(x, y) \rightarrow \exists z \text{Emp}(x, y, z) \quad \sigma_1$$
$$\text{EmpFax}(x, z) \rightarrow \exists y \text{Emp}(x, y, z) \quad \sigma_2$$

$$\Sigma_t: \quad \text{Emp}(x, y, z) \land \text{Emp}(x, u, v) \rightarrow y = u \land z = v \quad \sigma_3$$

If the source instance $I$ is $\{s_1: \text{EmpPhone}(\text{Mary}, \text{p123}), s_2: \text{EmpFax}(\text{Mary}, \text{f567})\}$, then a solution $J$ is a single tuple $t: \text{Emp}(\text{Mary}, \text{p123}, \text{f567})$. There are two routes for $t$: $s_1 \xrightarrow{\sigma_1} t$, and $s_2 \xrightarrow{\sigma_2} t$. We omit the homomorphisms in the routes and write $s_i$ instead of $(I, \emptyset)$ and $t$ instead of $(I, \{t\})$. In either case, the route makes an assumption about the values taken by the existentially-quantified variables. The variables $z$ and $y$ are assumed to be “f567” and “p123”, respectively. Observe that the egd $\sigma_3$ is not used in the routes.

A meaningful addition of egds to routes would necessitate an operational view of the exchange. An egd asserts that under certain conditions, some values should be equated with some other values. The information on which values change due to an egd is completely lost, unless additional information about the transformation is kept during the exchange or, sophisticated reasoning is applied to infer which values have been equated due to an egd. Our definition of routes allows one to reason about the specification of a data exchange at the level of the specification (i.e., schema mappings), without being tied to any operational semantics that is associated with schema mappings. In the future, we would also like to explore an alternative
definition of routes based on the chase procedure for data exchange [44] and explore algorithms for computing routes for this definition.

3.2.1 Computing All Routes

This section describes an algorithm for computing all routes for a given set of tuples $J_s \subseteq J$, where $J$ is any solution for a source instance $I$ under the schema mapping $M$. The algorithm works for any solution, and it is not tied to the solutions generated with any particular schema mapping system (e.g., Clio, or RDA). The algorithm constructs a route forest, in polynomial time in the size of $I$, $J$ and $J_s$, that concisely represents all routes. We characterize what “all routes” means. More specifically, the main result of this section (Theorem 3.2.5) states that every minimal route for a set of target tuples is, essentially, represented in this route forest. Intuitively, a minimal route for a set of target tuples is a route where none of its satisfaction steps can be removed and the result still forms a route for the set of target tuples.

The algorithm for computing all routes is shown below. It makes use of a reverse chase procedure which is shown in Figure 3.3.

---

**Algorithm ComputeAllRoutes$_M(I,J,J_s)$**

**Input:** A source instance $I$, a solution $J$ for $I$ under $M$ and a set of tuples $J_s \subseteq J$.

**Output:** A route forest for $J_s$.

Global data structures:

- A set of ACTIVETUPLES that contains tuples for which the algorithm has attempted to find all routes. Initially, this set is empty.
**FindAllRoutes**($J_s$)

For every tuple $t$ in $J_s$, if $t$ is not in ACTIVETUPLES, then:

1. Add $t$ to ACTIVETUPLES.

2. For every $s$-$t$ tgd $\sigma$ and assignment $h$ such that $h$ is a possible assignment returned by the procedure $\text{findHom}(I, J, t, \sigma)$
   
   (a) Add $(\sigma, h)$ as a branch under $t$.

3. For every target tgd $\sigma$ and assignment $h$ such that $h$ is a possible assignment returned by procedure $\text{findHom}(I, J, t, \sigma)$
   
   (a) Add $(\sigma, h)$ as a branch under $t$.
   
   (b) $\text{FindAllRoutes}(\text{LHS}(h(\sigma)))$.

Return the constructed route forest for $J_s$.

Intuitively, for every tuple $t$ encountered during the construction of the route forest, the algorithm considers every possible $\sigma$ and $h$ such that $h$ is a possible assignment returned by $\text{findHom}(I, J, t, \sigma)$. Conceptually, this corresponds to all possible $(\sigma, h)$ pairs under the tuple $t$ in the route forest. In other words, the algorithm explores all possibilities of witnessing $t$.

Next, we illustrate the $\text{findHom}$ procedure with an example, and examine $\text{ComputeAllRoutes}$ in more detail.

Suppose $\text{findHom}(I, J, t_1, m_1)$ in invoked, where $I$, $J$, $t_1$ and $m_1$ are from Figure 3.1 and Figure 3.1. By “matching” $t_1$ with the atom Accounts($cn, l, s$) of $m_1$, the assignment $v_1$ defined in Step 1 of $\text{findHom}$ is $\{cn \mapsto 6689, l \mapsto 15K, s \mapsto 434\}$. When $v_1$ is applied to the left-hand-side (LHS) of $m_1$, the partially instantiated relational atom Cards(6689, 15K, 434, $n$, ...
findHom(I, J, t, σ)

Input: A source instance I, a solution J for I under M, a tuple t ∈ J of the form R(a), and a tgd σ in Σst ∪ Σt of the form ∀x φ(x) → ∃y ψ(x, y).

Output: An assignment h such that h(φ(x)) ⊆ K, h(ψ(x, y)) ⊆ J and t ∈ h(ψ(x, y)).

1. Let R(z) be a relational atom of ψ(x, y). If no such relational atom can be found, return failure. Otherwise, let v1 be a mapping that assigns the ith variable of z to the ith value of a in R(a). If v1 assigns a variable z to two different values under this mapping scheme, repeat Step 1 with a different relational atom R(z) of ψ(x, y). If no other relational atom R(z) of ψ(x, y) can be found, return failure.

2. Let v2 be an assignment of variables in v1(φ(x)) to values in K so that v2(v1(φ(x))) ⊆ K, where K denotes I if σ is a s-t tgd, and K denotes J if σ is a target tgd.

3. Let v3 be an assignment of variables in v2(v1(ψ(x, y))) to values in J so that v3(v2(v1(ψ(x, y)))) ⊆ J.

4. Return v1 ∪ v2 ∪ v3.

Figure 3.3: The findHom procedure.

m, sal, loc) is obtained. Hence, the assignment v2 obtained in Step 2 is \{n → J. Long, m → Smith, sal → 50K, loc → Seattle\}. With v1 ∪ v2, the LHS of m1 corresponds to the tuple s1 in the Cards relation. The RHS of m1 is the conjunction of tuples Accounts(6689,15K,434) and Clients(434,Smith,Smith,50K,A). Hence, Step 3 of findHom returns v3 as \{A → A1\}. The algorithm then returns v1 ∪ v2 ∪ v3 in Step 4. In general, there are many possible assignments for v1, v2 and v3 for a tgd σ. The algorithm looks for one combination of v1, v2 and v3 that works. In our implementation of the algorithm, the evaluation for v2 and v3 is pushed to the database, as we shall explain in Section 3.4.
Example 3.2.4 Let $\mathcal{M}$ be a schema mapping where $\Sigma_{st}$ and $\Sigma_{t}$ consists of the following tgd's.

$$
\begin{align*}
\Sigma_{st} & : & S_1(x) \rightarrow T_1(x) & \sigma_1 \\
& & S_2(x) \rightarrow T_2(x) & \sigma_2 \\
\Sigma_{t} & : & T_2(x) \rightarrow T_3(x) & \sigma_3 \\
& & T_3(x) \rightarrow T_4(x) & \sigma_4 \\
& & T_4(x) \land T_1(x) \rightarrow T_5(x) & \sigma_5 \\
& & T_4(x) \land T_6(x) \rightarrow T_7(x) & \sigma_6 \\
& & T_5(x) \rightarrow T_3(x) & \sigma_7 \\
& & T_5(x) \rightarrow T_6(x) & \sigma_8 \\
\end{align*}
$$

It is easy to see that $\mathcal{M}$ is recursive, since the graph of dependencies in $\Sigma_{t}$ contains a cycle involving $T_3$, $T_4$, and $T_5$. Let the source instance $I$ consist of two tuples $S_1(a)$ and $S_2(a)$. A solution $J$ for $I$ under $\mathcal{M}$ consists of tuples $T_1(a), ..., T_7(a)$. Suppose we wish to compute all routes for $T_7(a)$. That is, $\text{ComputeAllRoutes}_{\mathcal{M}}(I, J, \{T_7(a)\})$ is invoked. The route forest (in this case, a tree) that is constructed by the algorithm is shown in Figure 3.4. (Please disregard the dotted branches for this example.) The order at which the branches are added to the forest and the steps involved in $\text{ComputeAllRoutes}$ are labeled as a pair beside the branches in the tree. For example, $(N2, S3)$ for the branch with $\sigma_4$ denotes that this is the second branch added in the construction and it is added by Step 3 of $\text{ComputeAllRoutes}$. In this example, $h$ is always $\{x \mapsto a\}$ and is omitted from the figure.

In the process of constructing a route tree for $T_7(a)$, Step 2 of $\text{ComputeAllRoutes}$ fails to add any branches to $T_7(a)$. However, Step 3 adds the $\sigma_6$ branch to $T_7(a)$ and continues the construction of the tree with $\text{FindAllRoutes}(\{T_4(a), T_6(a)\})$. Finding a route for $T_4(a)$ leads to the tuple $T_3(a)$. There are two branches, $\sigma_7$ and $\sigma_3$, for $T_3(a)$. In this computation, the $\sigma_7$ branch was explored before $\sigma_3$ and eventually, the $\sigma_7$ branch will cause $\text{FindAllRoutes}(\{T_4(a), T_1(a)\})$ to be invoked. However, since $T_4(a)$ belongs to ACTIVETUPLES at this point, the branches for $T_4(a)$ are not explored at this node. Similarly, there are no branches...
under the tuple $T_5(a)$ under the $\sigma_8$ branch because $T_5(a)$ is an active tuple at this point. Intuitively, the branches for an active tuple $t$ are added only when $t$ is first encountered during the construction of the forest.

Obviously, the resulting forest that is constructed is not unique. For instance, if $T_6(a)$ was selected to be explored before $T_4(a)$ in $\text{FindAllRoutes}(\{T_4(a), T_6(a)\})$, the constructed tree will be different from Figure 3.4. It is also easy to observe that $\text{ComputeAllRoutes}$ terminates since for each tuple $t$, there are only finitely many branches to add under $t$ in Steps 2 and 3. Furthermore, due to ACTIVETUPLES, the branches of an active tuple are only explored at one place in the forest. Hence, $\text{ComputeAllRoutes}$ runs in polynomial time in the size of $I$, $J$ and $J_s$ since there can only be polynomially many branches under each tuple.
Proposition 3.2.1 Let $M$ be a schema mapping. Let $I$ be a source instance, $J$ be a solution for $I$ under $M$ and $J_s \subseteq J$. Then, ComputeAllRoutes$_M(I,J,J_s)$ executes in polynomial time in the size of $I$, $J$ and $J_s$.

**Proof.** Let $n$ denote the number of distinct values in $I$, $J$ and $J_s$. There can be at most $k \cdot n^c$ branches (i.e., $(\sigma, h)$ pairs) for a tuple $t$, where $c$ denotes the maximum number of distinct variables in any tgd in $\Sigma_s \cup \Sigma_t$ and $k$ denotes the number of tgds in $\Sigma_s \cup \Sigma_t$. Since $M$ is fixed, $c$ and $k$ are constants. Due to ACTIVETUPLES, every branch of each tuple is explored at most once. Therefore, there can be at most $|J| \cdot k \cdot n^c$ branches to explore, where $|J|$ denotes the number of tuples in $J$. Hence, the algorithm terminates in polynomial time.

From the tree in Figure 3.4, it is easy to see that a route, $R_1$, for $T_7(a)$ is:

$$R_1: I \xrightarrow{\sigma_2} I, T_2 \xrightarrow{\sigma_3} I, T_2, T_3 \xrightarrow{\sigma_4} I, T_2, T_3, T_4 \xrightarrow{\sigma_1} I, T_1, ..., T_4$$

$$\xrightarrow{\sigma_5} I, T_1, ..., T_5 \xrightarrow{\sigma_8} I, T_1, ..., T_6 \xrightarrow{\sigma_6} I, T_1, ..., T_7$$

For conciseness, $T_i(a)$ is written as $T_i$ in the route $R_1$ above. If there is another s-t tgd $\sigma_9 : S_3(x) \rightarrow T_5(x)$ and assuming that $I$ also contains the source tuple $S_3(a)$, then there would be another branch under the tuple $T_5(a)$ under $\sigma_7$. (See the leftmost dotted branch in Figure 3.4.) This would mean there is another route for $T_7(a)$:

$$R_2: I \xrightarrow{\sigma_9} I, T_5 \xrightarrow{\sigma_7} I, T_5, T_3 \xrightarrow{\sigma_4} I, T_5, T_3, T_4 \xrightarrow{\sigma_8} I, T_5, T_3, T_4, T_6 \xrightarrow{\sigma_6} I, T_5, T_3, T_4, T_6, T_7$$

In the route $R_2$, the tuple $T_1(a)$ is “by-passed” since it is now possible to witness $T_5(a)$ directly with the s-t tgd $\sigma_9$. 

116
**Completeness of the route forest in representing all routes** The route forest generated by ComputeAllRoutes is complete in the sense that every minimal route for $J_s$ can, essentially, be found in this route forest. More specifically, every minimal route for $J_s$ is represented by one of the routes which are naively generated from this forest. The procedure for naively generating routes for $J_s$ is shown below. It finds the set of all routes for every tuple $t$ in $J_s$ and takes a cartesian product of these sets of routes in the last step.

---

**NaivePrint$_F$(J$_s$)**

Denote by $F$ the route forest returned by ComputeAllRoutes$_M$(I,J,J$_s$), where $I$ is a source instance, $J$ is a solution for I under $M$ and $J_s \subseteq J$.

**Input:** A set of tuples $J_s$ where every tuple in $J_s$ occurs in $F$. Assumes a global stack ANCESTORS which is initially empty.

**Output:** A set of all routes for $J_s$.

For every tuple $t$ in $J_s$

1. Push $t$ into ANCESTORS. Goto any $t$ in $F$.

2. Let $L_1$ denote the set of all $(\sigma, h)$ branches under $t$ such that $\sigma$ is a s-t tgd.

3. Let $L_2$ denote the set of all $(\sigma, h)$ branches under $t$ such that $\sigma$ is a target tgd and every tuple in LHS($h(\sigma)$) does not occur in ANCESTORS.

4. Let $L_3 = \emptyset$.

5. For every $(\sigma, h)$ in $L_2$

   (a) Let $L'$ denote NaivePrint$_F$(LHS($h(\sigma)$)).

   (b) Append $(\sigma, h)$ to every element in $L'$.
(c) $L_3 = L_3 \cup L'$.

6. Let $L(t)$ be $L_1 \cup L_3$.

7. Pop ANCESTORS.

Return $L(t_1) \times \ldots \times L(t_k)$, where $J_s = \{t_1, \ldots, t_k\}$.

In Step 1, the search for routes is allowed to start from any occurrence of $t$ in $F$. Even though there may be many occurrences of $t$ in $F$, we assume that every other occurrence of $t$ has a link to the first $t$ in $F$ where the branches of $t$ are explored. For example, $T_4(a)$ under the $\sigma_5$ branch has a reference to $T_4(a)$ under the $\sigma_6$ branch. NaivePrint on $T_7(a)$ will produce a route $\sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_4$ for $T_4(a)$ and a route $\sigma_2 \rightarrow \sigma_3 \rightarrow \sigma_1 \rightarrow \sigma_5 \rightarrow \sigma_8$ for $T_6(a)$. For conciseness, only the tgd's involved are listed in these routes. Hence, the route produced for $T_7(a)$ is as shown below.

$$R_3 : \sigma_2, \sigma_3, \sigma_4, \sigma_2, \sigma_3, \sigma_4, \sigma_1, \sigma_5, \sigma_8, \sigma_6$$

Obviously, this route contains some redundant satisfaction steps. Intuitively, given a set of target tuples $J_s$ and a route $R$ for $J_s$, a satisfaction step of $R$ is redundant if the result of removing this step from $R$ is still a route for $J_s$. We say that a route is minimal if it does not contain any redundant satisfaction steps. It is easy to see that a minimal route for $T_7(a)$ is in fact $R_1$. Although $R_3$ is not a minimal route, it has the same set of satisfaction steps as the route $R_1$.

To compare routes based on the satisfaction steps they use, rather than the order in which the satisfaction steps are used, we introduce an interesting concept called the stratified
interpretation of routes. To stratify a route, we make use of the concept of the rank of a tuple in a route. Intuitively, every tuple is associated with a unique rank in a route. Source tuples have rank 0 and a tuple $t$ has rank $k$ in a route if for some satisfaction step in the route that involves $\sigma$ and $h$, (1) $t$ occurs in RHS($h(\sigma)$), (2) the maximum rank of tuples in LHS($h(\sigma)$) is $k - 1$, and (3) $t$ is not of a lower rank. The stratified interpretation of a route $R$, denoted as strat($R$), partitions the pairs $(\sigma, h)$ in $R$ into blocks. A satisfaction step $(\sigma, h)$ of a route belongs to rank 1 if LHS($h(\sigma)$) consists of only source tuples, and it belongs to rank $k$ if the maximum rank of tuples in LHS($h(\sigma)$) is $k - 1$. The rank of a route is the number of blocks in the stratified interpretation of the route. For example, $R_1$ and $R_3$ have the same stratified interpretation shown below and they both have rank 6.

<table>
<thead>
<tr>
<th>Rank 1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma_1, \sigma_2$</td>
<td>$\sigma_3$</td>
<td>$\sigma_4$</td>
<td>$\sigma_5$</td>
<td>$\sigma_8$</td>
<td>$\sigma_6$</td>
</tr>
</tbody>
</table>

Two routes $R$ and $R'$ have the same stratified interpretations, denoted as strat($R$) = strat($R'$), if for every block of rank $i$ in strat($R$), every $(\sigma, h)$ in this block can be found in the corresponding $i$th block of strat($R'$) and vice versa. It is easy to see that if two routes $R$ and $R'$ have the same stratified interpretation, then they use the same set of satisfaction steps. (The order in which these satisfaction steps occur in $R$ and respectively $R'$, might be different). Hence, $R$ and $R'$ are essentially the same, modulo the ordering of their satisfaction steps. We have introduced the notion of stratified routes because it is more easily understandable, when compared to a sequence of satisfaction steps. Furthermore, the stratified interpretation of routes arises naturally in the proof of Theorem 3.2.5, the main result of this section which states the completeness of the route forest in representing all routes. Specifically, Theorem 3.2.5
states that for any $J_s \subseteq J$, the forest $F$ that is constructed by $\text{ComputeAllRoutes}_M(I, J, J_s)$ contains all routes for $J_s$, in the sense that every minimal route for $J_s$ has the same stratified interpretation as one of the routes produced by $\text{NaivePrint}_F(J_s)$. The proof of Theorem 3.2.5 makes use of Lemma 3.2.1 stated below.

**Lemma 3.2.1** Let $M$ be a schema mapping. Let $I$ be a source instance and $J$ be a solution for $I$ under $M$. Let $J_s \subseteq J$ and let $F$ denote the route forest of $\text{ComputeAllRoutes}_M(I, J, J_s)$. if $R$ is a minimal route for a tuple $t$ that occurs in $F$, then there exists a route $R'$ that is returned by $\text{NaivePrint}_F(\{t\})$ with the property that $\text{strat}(R) = \text{strat}(R')$.

**Proof.** The proof of the lemma is by induction on the rank $n$ of $R$.

**Base case:** $\text{rank}(R) = 1$. Let $t$ be a tuple that appears in $F$ such that there exists a minimal route $R$ for $t$ with rank 1. This means that $R$ must consist of a single satisfaction step $(\sigma, h)$, where $\sigma$ is a s-t tgd. Clearly, Step 2 of $\text{ComputeAllRoutes}$ creates the branch $(\sigma, h)$ under $t$ in $F$, more precisely, where the branches of $t$ are first explored. Step 2 of $\text{NaivePrint}_F(\{t\})$ adds the branch $(\sigma, h)$ to $L(t)$. Hence, $\text{NaivePrint}_F(\{t\})$ returns the route $R'$ which is identical to $R$, and so $\text{strat}(R') = \text{strat}(R)$ trivially.

**Inductive hypothesis:** $\text{rank}(R) \leq n$. Let $t$ be a tuple which occurs in $F$ such that there exist a minimal route $R$ for $t$ such that $\text{rank}(R) \leq n$. Then there exists a route $R'$ that is returned by $\text{NaivePrint}_F(\{t\})$ with the property that $\text{strat}(R) = \text{strat}(R')$.

Let $t$ be a tuple in $F$ such that there exist a minimal route $R$ for $t$ of rank $n + 1$. We show that there exists a route $R'$ that is returned by $\text{NaivePrint}_F(\{t\})$ with the property that $\text{strat}(R) = \text{strat}(R')$. Since $R$ is minimal route for $t$, it must be that $t$ is witnessed in the last
satisfaction step of $R$ with a target tgd, since $\text{rank}(t) \geq 1$. Denote this satisfaction step as $(\sigma, h)$, where $\sigma$ is a target tgd. Let $M$ denote $R$ without this last satisfaction step. Clearly, $M$ is a minimal route for $\text{LHS}(h(\sigma))$ of rank $n$.

Let $\{t_1, \ldots, t_k\}$ be the tuples in $\text{LHS}(h(\sigma))$ and consider the node of $t$ in $F$ where the branch $(\sigma, h)$ was added under $t$ during $\text{ComputeAllRoutes}(I, J, J_a)$. The branch $(\sigma, h)$ must be added to $L_2$ in Step 3 of $\text{NaivePrint}_F(\{t\})$, since at this point $\text{ANCESTORS} = \{t\}$ and $t$ is not in $\text{LHS}(h(\sigma))$.

Since $M$ is a minimal route for $\{t_1, \ldots, t_k\}$, it must be possible to extract routes $M_1, \ldots, M_k$ from $M$ such that $M_i$ is a minimal route for $t_i$, $1 \leq i \leq k$, and $\text{strat}(M_1, \ldots, M_k) = \text{strat}(M)$. Above, $M_1, \ldots, M_k$ denotes the route obtained by repeatedly appending to $M_1$ the satisfaction steps of $M_2$ to $M_k$, respectively. Otherwise, there must be a satisfaction step $s$ in $M$ that is not in the union of all steps in $M_1$ to $M_k$. Hence, $M$ without $s$ is also a route for $\{t_1, \ldots, t_k\}$, thus contradicting the minimality of $M$.

Since $M$ is a minimal route for $\{t_1, \ldots, t_k\}$ of rank $n$, it follows that each $M_i$ is of rank $j \leq n$. Hence, by induction hypothesis, there must be a route $M'_i$ returned by $\text{NaivePrint}_F(\{t_i\})$ with the property that $\text{strat}(M_i) = \text{strat}(M'_i)$. Note that by the construction of $F$, every $t_i$ must occur in $F$. Since the for-loop of $\text{NaivePrint}_F(\{t_1, \ldots, t_k\})$ will construct $L(t_i)$, $1 \leq i \leq k$, where $L(t_i)$ denotes the set of all routes for $t_i$ and $M'_i$ is a route in $L(t_i)$, we have that $\text{NaivePrint}$ on $\{t_1, \ldots, t_k\}$ will return $M' = M'_1, \ldots, M'_k$ as one of the routes in $L(t_1) \times \ldots \times L(t_k)$ in the last step. Hence, Step 5(b) of $\text{NaivePrint}_F(\{t\})$ will append the satisfaction step $(\sigma, h)$ to $M'$ and add the resulting route $R' = M'.(\sigma, h)$ to $L(t)$, where $M'.(\sigma, h)$ denotes the result of appending the satisfaction step $(\sigma, h)$ to $M'$. Since $\text{strat}(M_i) = \text{strat}(M'_i)$, if follows that
strat(M) = strat(M'). Hence, strat(R) = strat(M.(σ,h)) = strat(M'.(σ,h)) = strat(R'), which
was to be shown.

Theorem 3.2.5 Let M be a schema mapping. Let I be a source instance and J be a solution
for I under M. Let J_s ⊆ J and let F denote the route forest of ComputeAllRoutes_M(I,J,J_s).
If R is a minimal route for J_s, then there exists a route R' in the result of NaivePrint_F(J_s) with
the property that strat(R) = strat(R').

Proof. Let F be the route forest that is constructed by ComputeAllRoutes_M(I, J, J_s) and let
{t_1, ..., t_k} be the set of tuples in J_s. Since R is a minimal route for J_s, it must be possible to
extract routes R_1, ..., R_k from R such that R_i is a minimal route for t_i and strat(R_1, ..., R_k) =
strat(R). By Lemma 3.2.1, there must be a route R'_i returned by NaivePrint_F({t_i}) with the
property that strat(R_i) = strat(R'_i). Note that by the construction of F, every t_i must occur in F.
Since the for-loop of NaivePrint on J_s will construct L(t_i), 1 ≤ i ≤ k, where L(t_i) denotes
the set of all routes for t_i and R'_i is a route in L(t_i), we have that NaivePrint on J_s will return
R'_1, ..., R'_k as one of the routes in L(t_1)x...xL(t_k) in the last step. Since strat(R_i) = strat(R'_i),
we have strat(R) = strat(R'_1, ..., R'_k).

Note that the route forest is a compact, polynomial-size representation of all routes,
even though there may be exponentially many minimal routes for J_s. We remark that the number
of minimal routes is always polynomial when the set of target dependencies Σ_t is acyclic (i.e.,
there are no cycles in the graph of Σ_t where nodes represent relation symbols and there is an
dge (R,S) whenever R and S appear in the LHS and, respectively, RHS of a tgd in Σ_t). It is
Infer($S$)

Repeat until $S = \emptyset$

1. Let $S = \emptyset$.

2. For every triple $(t', \sigma', h')$ in UNPROVEN

   (a) If LHS($h'(\sigma')$) consists of only proven tuples, then

      (i) Add $t'$ to $S$.

      (ii) Remove $(t', \sigma', h')$ from UNPROVEN.

      (iii) Append $(\sigma', h')$ to $G$; Mark $t'$ as proven.

---

**Figure 3.5:** The Infer procedure used by ComputeOneRoute.

only when the schema mapping is recursive that we may have exponentially many routes. We say that a schema mapping $\mathcal{M} = (S, T, \Sigma_{st}, \Sigma_t)$ is recursive if the graph of $\Sigma_t$ contains a cycle.

The next example illustrates such a schema mapping.

**Example 3.2.6** Let $\mathcal{M}$ be a schema mapping where $\Sigma_{st}$ and $\Sigma_t$ consists of the following tgd.

\[
\Sigma_{st} : \begin{align*}
R_1(x, y) & \rightarrow S(x, y) \\
R_2(x, y) & \rightarrow S(x, y) \\
R_3(x, y) & \rightarrow T(x, y)
\end{align*}
\]

\[
\Sigma_t : S(x, y) \land S(y, z) \land T(x, y) \rightarrow T(y, z)
\]

(\sigma_1, \sigma_2, \sigma_3, \sigma_4)

Note that the graph of $\Sigma_t$ has a cycle involving $T$. Consider a source instance $I$ consisting of tuples $R_1(1, 2), R_1(2, 3), \ldots, R_1(n-1, n), R_2(1, 2), R_2(2, 3), \ldots, R_2(n-1, n)$, and $R_3(1, 2)$. A solution $J$ for $I$ under $\mathcal{M}$ consists of tuples $S(1, 2), S(2, 3), \ldots, S(n-1, n)$, and $T(1, 2), T(2, 3), \ldots, T(n-1, n)$. A route forest (a tree in this case) computed for the tuple
Figure 3.6: A route tree for $T(n-1, n)$ embedding exponential many minimal routes.

$T(n-1,n)$ is illustrated in Figure 3.6. Note that the branches under $S(2, 3)$ are not explored at the bottom level of the forest, since they have been explored one level above. The same holds for $S(3, 4)$.

It is easy to see that there are 4 minimal routes for $T(2, 3)$, since each of $S(1, 2)$ and $S(2, 3)$ has two possible routes with $\sigma_1$ and $\sigma_2$, respectively. Furthermore, since there are 2 minimal routes for $S(3, 4)$, we can count $2 \times 4 = 8$ minimal routes for $T(3, 4)$, and so on. A simple inductive argument shows that there are $2^{n-1}$ minimal routes for $T(n-1, n)$. At the same time, the size of the route forest is polynomial in $n$, since the depth of the forest is $n$, and there are at most 5 branches at each level.

The experimental results in Section 3.5 indicate that it may be expensive to construct
the route forest in general. Hence, a natural question is whether it is possible to produce one route fast and leverage the “debugging-time” of the user to produce other routes as needed.

### 3.2.2 Computing One Route

In debugging, it would be useful to have the alternative feature to derive and display one route fast and display other routes, as needed. The experimental results in Section 3.5 justify that in most cases, it is much faster to compute one route than compute all routes. In general however, it is valuable to incorporate both features for a debugger. In some cases, the user may be satisfied with one route, which is faster to compute than computing all routes. It is also useful, however, to be able to determine all routes whenever desired.

The algorithm **ComputeOneRoute** which computes a route for a given set of tuples $J_s \subseteq J$, where $J$ is any solution for $I$ under the schema mapping $\mathcal{M}$, is described below. The algorithm makes use of the **Infer** procedure shown in Figure 3.5. It also uses the **findHom** procedure shown in Figure 3.3 and described in Section 3.2.2.

---

**Algorithm** ComputeOneRoute$_{\mathcal{M}}(I,J,J_s)$

**Input:** A source instance $I$, a solution $J$ for $I$ under $\mathcal{M}$ and a set of tuples $J_s \subseteq J$.

**Output:** A route for $J_s$.

Global data structures:

- A set of ACTIVETUPLES that contains tuples for which the algorithm has attempted to find a route. Initially, this set is empty.
- A set UNPROVEN that contains unproven triples, initially empty.
- Every tuple has a status proven or unproven.
• A sequence of pairs $G$ used to contain the route, initially empty.

FindRoute($J_s$)

For every tuple $t$ in $J_s$, if $t$ is not in ACTIVETUPLES, then:

1. Add $t$ to ACTIVETUPLES.

2. If findHom($I$, $J$, $t$, $\sigma$) returns $h$ for some s-t tgd $\sigma$, then

   (a) Append ($\sigma$, $h$) to $G$; Mark $t$ as proven.

   (b) Infer({$t$}).

   (c) Continue with the next iteration of For-Loop.

3. If findHom($I$, $J$, $t$, $\sigma$) returns $h$ for some target tgd $\sigma$, then

   (a) If LHS($h(\sigma)$) consists of only proven tuples, then

      (i) Append ($\sigma$, $h$) to $G$; Mark $t$ as proven.

      (ii) Infer({$t$}).

   Else

      (iii) Add ($t$, $\sigma$, $h$) to UNPROVEN.

      (iv) FindRoute(LHS($h(\sigma)$)).

   (v) If ($t$, $\sigma$, $h$) is UNPROVEN, continue with Step 3 searching for another ($\sigma$, $h$) pair for $t$.

Return $G$.

Next, the algorithm ComputeOneRoute is examined in detail by means of an example. A comparison with ComputeAllRoutes is described shortly.

**Example 3.2.7** Let $\mathcal{M}$, $I$ and $J$ be the schema mapping, source and target instances given in Example 3.2.4. Suppose a route for $T_7(a)$ is sought. With ComputeOneRoute$_{\mathcal{M}}(I, J$, 126
The same route tree of Figure 3.4 is obtained. (Please disregard the dotted branches.) The computation that occurs during the construction, however, is different. With the tuple \( T_7(a) \), ComputeOneRoute fails to find a s-t tgd in Step 2 for \( T_7(a) \). Hence, it proceeds to Step 3 and succeeds with reverse chasing \( T_7(a) \) with \( \sigma_6 \). (As before, \( h \) is omitted as it is always \( \{x \mapsto a\} \) in this example.) Consequently, \( \text{FindRoute}(\{T_4(a), T_6(a)\}) \) is invoked. Similarly, for \( T_4(a) \), reverse chase succeeds with \( \sigma_4 \). For \( T_5(a) \), reverse chase succeeds with \( \sigma_5 \). In the branch with \( \sigma_5 \), \( \text{FindRoute}(\{T_4(a), T_1(a)\}) \) is invoked. Since \( T_4(a) \) occurs in \text{ACTIVETULES} \text{, the For-Loop for FindRoute for T_4(a) is not executed. Instead, FindRoute continues with the tuple T_1(a) and succeeds with a s-t tgd \( \sigma_1 \). Since reverse chase does not succeed with other tgds on \( T_5(a) \), the algorithm returns to \( T_3(a) \). It happens that, so far, the computation resembles ComputeAllRoutes. Continuing from \( T_3(a) \), the algorithm succeeds in witnessing \( T_3(a) \) with \( \sigma_3 \) and \( \sigma_2 \). At \( \sigma_2 \), the set \text{UNPROVEN} is \( \{\sigma_6, \sigma_4, \sigma_7, \sigma_5, \sigma_3\} \), and \( G \) is the sequence \( [\sigma_1, \sigma_2] \). When \( \text{Infer}(\{T_2(a)\}) \) is invoked in Step 2(b), the algorithm will deduce that \( T_3(a), T_4(a), T_5(a) \) and \( T_3(a) \) are proven, in this order, and \( G \) is now \( [\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_7] \). After this, the algorithm returns to the branch \( \sigma_6 \) and attempts to find a route for \( T_6(a) \) next. It succeeds with \( \sigma_8 \) because \( T_5(a) \) is already proven and it will infer that \( T_7(a) \) is proven with \( \text{Infer}(\{T_6(a)\}) \) in Step 3(a-ii) of ComputeOneRoute. The algorithm successfully terminates and returns \( [\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_7, \sigma_8, \sigma_6] \).

We note at this point that our implementation of ComputeOneRoute in SPIDER is an optimization of the algorithm described above. If the reverse chase step for a tuple \( t \) is successful with some tgd \( \sigma \) in Step 2 or Step 3 of the algorithm, SPIDER concludes that all the
target tuples produced by \( \sigma \) (and not only \( t \)) are proven. Hence, SPIDER may avoid performing redundant reverse chase steps with the rest of the tuples. In this section, however, we shall assume that only \( t \) is proven, for simplicity in reasoning about ComputeOneRoute.

**Comparison between Computing All Routes and Computing One Route** In ComputeOneRoute, the algorithm searches for one successful branch under a tuple \( t \) to find a route for \( t \). In ComputeAllRoutes, however, all branches are searched, regardless of whether a route for \( t \) has already been found. To make a better contrast, suppose \( \Sigma_{st} \) contains another s-t tgd \( \sigma_9 : S_3(x) \rightarrow T_5(x) \) and additionally, there is a source tuple \( S_3(a) \) in \( I \). Then, for the tuple \( T_5(a) \) under the branch \( \sigma_7 \) in Figure 3.4, only the branch \( \sigma_9 \) will be considered. This is because the algorithm considers s-t tgds before target tgds (Step 2 of ComputeOneRoute). Since a route for \( T_5(a) \) can be found with the s-t tgd \( \sigma_9 \), the branch with \( \sigma_5 \) will not be explored.

The second difference is that the result returned by ComputeOneRoute is a sequence of \((\sigma, h)\) pairs that represents the route that is found for \( J_s \), even though a route forest is constructed during the computation. These \((\sigma, h)\) pairs are collected during the construction of the forest in ComputeOneRoute.

The third difference is that in ComputeOneRoute utilizes an Infer procedure to infer proven tuples as the forest is constructed. This procedure is needed for the correctness of the algorithm. To illustrate, consider Example 3.2.7 and the execution of ComputeOneRoute for \( T_7(a) \) without Infer. At the \( \sigma_2 \) branch, no inference will be made. Although it is possible to conclude that \( T_3(a) \) and \( T_4(a) \) are proven as the algorithm returns along the branches \( \sigma_3 \) and \( \sigma_4 \) respectively, it is not possible to conclude that \( T_5(a) \) is proven. Hence, without Infer, the status of \( T_5(a) \) is unknown at the point when \( \sigma_8 \) is used and because \( T_5(a) \) is in ACTIVETUPLES,
the branches under $T_5(a)$ are not explored. The algorithm therefore terminates with a partial route for $T_7(a)$, which is incorrect. One might argue that the ACTIVETUPLES restriction could be removed to allow the branches under $T_5(a)$ to be explored again. In this case, since both $T_4(a)$ and $T_1(a)$ are proven by the time the branches of $T_5(a)$ are explored under $\sigma_8$, the algorithms concludes that $T_5(a)$ is proven and so the algorithm terminates with a route for $T_7(a)$. However, without the ACTIVETUPLES restriction, there might be many unnecessary explorations. To illustrate, suppose the schema mapping of Example 3.2.7 has an additional target $\text{tg}_{10} : T_5(x) \land T_8(y) \rightarrow T_3(x)$ and the target instance $J$ consists of $n$ additional tuples $T_8(b_1), ..., T_8(b_n)$. Observe that $J$ is still a solution for $I$ under $M$ with these $n$ additional tuples. When $T_3(a)$ is encountered during the construction of a route for $T_7(a)$, it may happen that the $\sigma_3$ branch is explored last. (Refer to $T_3(a)$ of Figure 3.4.) Hence, the branches of $T_5(a)$ are repeatedly explored along the branches $\sigma_7, (\sigma_{10}, h_1), ..., (\sigma_{10}, h_n)$, where $h_i = \{x \rightarrow a, y \rightarrow b_i\}$. The repeated exploration of the branches of $T_5(a)$ in this case is unnecessary.

Furthermore, the ACTIVETUPLES restriction makes the running time analysis simpler. As in ComputeAllRoutes, since every $(\sigma, h)$ pair for a tuple $t$ occurs at most once in the route forest, there are at most polynomially many branches in this forest that is constructed by ComputeOneRoute. In ComputeOneRoute, however, it is necessary to also reason about the running time of Infer.

**Proposition 3.2.2** Let $M = (S, T, \Sigma_{st}, \Sigma_t)$. Let $I$ be a source instance, $J$ be a solution for $I$ under $M$ and $J_s \subseteq J$. Then, ComputeOneRoute$_M(I, J, J_s)$ executes in polynomial time in the sizes of $I$, $J$ and $J_s$. 

---

129
Proof. From Proposition 3.2.1, the number of branches to explore is at most \( k \cdot n^c \), where \( k \) denotes the number of tgds in \( \Sigma_{st} \cup \Sigma_t \), \( n \) denotes the number of distinct values in \( I \), \( J \) and \( J_s \), and \( c \) denotes the maximum number of distinct variables in any tgd in \( \Sigma_{st} \cup \Sigma_t \). The running time of \texttt{Inf} is analyzed next. Since \( M \) is fixed, it takes \( m \cdot |J| \) amount of time to check if \( \text{LHS}(h'(\sigma')) \) consists of only proven tuples (Step 3(a)), where \( m \) is the maximum number of relational atoms in the LHS of any tgd, and \( |J| \) denotes the number of tuples in \( J \). Steps 3(a-i) and 3(a-iii) take a constant amount of time, and Step 3(a-ii) takes at most \( k \cdot n^c \) time. Since there are at most \( k \cdot n^c \) UNPROVEN branches, \texttt{Inf} takes \( k \cdot n^c \cdot (m \cdot |J| + k \cdot n^c) \) time. Denote this expression as \( Q(n) \). At worst, the For-Loop is invoked once for every proven tuple. Since there can be at most \( |J| \) invocations of the for-loop, the running time of \texttt{Inf} during the entire computation of \texttt{ComputeOneRoute} is \( |J| \cdot Q(n) \).

The running time analysis in Proposition 3.2.2 is based on the sizes of \( I \), \( J \) and \( J_s \). The default behavior of SPIDER, however, is to operate in conjunction with an existing schema mapping system (e.g., Clio, or RDA). In this case, SPIDER uses the solution \( J \) that is generated by this system on a given source instance \( I \) under a schema mapping \( M \). Therefore, a natural question is whether the polynomial time results of Propositions 3.2.1 and 3.2.2 hold when analyzed against the size of the source instance \( I \). It is easy to show that if \( J \) is polynomial in the size of \( I \), then both route algorithms run in polynomial time in the size of \( I \). It was shown in [44] that for a relational-to-relational schema mapping, if \( \Sigma_t \) is the union of a finite set of egds with a weakly acyclic finite set of tgds, then given a source instance \( I \), a solution \( J \) for \( I \) in a fixed schema mapping can be computed in polynomial time in the size of \( I \). (The notion of
weak acyclicity was formulated by A. Deutsch and L. Popa and independently used in [40] and [44].) In this case, the route algorithms run in polynomial time in the size of $I$ as well.

In the rest of this section, we shall show that ComputeOneRoute is complete for finding one route. The main result of this section is Theorem 3.2.8, which states that if there is a route for $J_s$, then ComputeOneRoute on $J_s$ will produce a route for $J_s$. In proving Theorem 3.2.8, we first prove Proposition 3.2.3 below which states that the result of ComputeOneRoute on $J_s$ is a route for every tuple that is marked as proven by the algorithm. For Theorem 3.2.8, we first show that for every tuple $t$ in $J_s$, if there is a route for $t$ then $t$ is marked as proven by the time the algorithm terminates. Since there is a route for $J_s$, it follows that there is a route for each tuple in $J_s$. Hence, by Proposition 3.2.3 we have that the result of ComputeOneRoute on $J_s$ is a route for each tuple in $J_s$. Therefore, the result of ComputeOneRoute on $J_s$ is also a route for $J_s$, which completes the proof of Theorem 3.2.8.

We next show that the result of ComputeOneRoute is a route for every tuple that is marked as proven during the execution of the algorithm.

**Proposition 3.2.3** Let $\mathcal{M}$ be a schema mapping, $I$ be a source instance and $J$ be a solution for $I$ under $\mathcal{M}$. Let $J_s \subseteq J$. For any tuple $t$ which has been marked as proven during the execution of ComputeOneRoute$_{\mathcal{M}}(I, J, J_s)$, the result of ComputeOneRoute$_{\mathcal{M}}(I, J, J_s)$ is a route for $t$.

**Proof.** Let $G$ be the global data structure which is used to hold the result of ComputeOneRoute$_{\mathcal{M}}(I, J, J_s)$. We prove the following stronger statement: At any Step $X$ during the execution of ComputeOneRoute$_{\mathcal{M}}(I, J, J_s)$, the content of $G$ is a route for the set of all tuples
which have been marked as proven by the algorithm so far. By the algorithm, it is easy to see
that no satisfaction step is ever removed from $G$. So if $G$ is a route for the set of proven tuples
at Step $X$, $G$ continues to be a route for these tuples when the algorithm terminates. It follows
immediately that the result of $\text{ComputeOneRoute}_{\mathcal{A}}(I, J, J_s)$ is a route for all tuples marked
as proven by the algorithm.

We next prove the above statement. Consider any Step $X$ during the execution of
$\text{ComputeOneRoute}_{\mathcal{A}}(I, J, J_s)$. Let $J_P$ be the set of target tuples in $J$ which have been
marked as proven by the algorithm up to Step $X$ of the execution. We shall show that the global
sequence $G$ constructed so far by the algorithm is a route for $J_P$. The proof is by induction
on the length of $G$ in Step $X$ of the algorithm. Note that the only steps when an element is
appended to $G$ are Steps 2(a) and 3(a-i) of $\text{ComputeOneRoute}$, and Step 2(a-iii) of $\text{Infer}$.
Also note that exactly one tuple is marked as proven in each of these steps. Hence, the length
of $G$ is equal to the size of $J_P$ at any execution step of $\text{ComputeOneRoute}_{\mathcal{A}}(I, J, J_s)$.

Base case: $G$ is of length 1. This means that $G$ consists of a single pair $(\sigma, h)$ and $J_P$ consists
of a single proven tuple $t$. Let $Y$ be the step in the execution of $\text{ComputeOneRoute}_{\mathcal{A}}(I, J, J_s)$ where $(\sigma, h)$ has been appended to $G$. We show next that $\sigma$ is a s-t tgd which witnesses $t
with homomorphism $h$. It follows that $(\sigma, h)$ has been appended to $G$ in Step 2(a) of $\text{Compute-
OneRoute}$. In other words, $Y$ must be Step 2(a) of $\text{ComputeOneRoute}$. Hence, after Step $Y$
is executed, $G$ is a route for $t$, and hence it is a route for $J_P$.

Suppose $\sigma$ is not a s-t tgd. Then $\sigma$ is a target tgd and there are two cases to analyze.

Case 1: $(\sigma, h)$ has been appended to $G$ in Step 3(a-i) of $\text{ComputeOneRoute}$. In other words,
$Y$ is Step 3(a-i) of $\text{ComputeOneRoute}$. Then it must be that the algorithm must have proven
some tuples prior to the execution of Step \( Y \). This means that \( J_P \) is non-empty at the time Step \( Y \) is executed, which contradicts the fact that \( J_P \) is a singleton set consisting of the tuple \( t \) after the execution of Step \( Y \).

**Case 2:** \((\sigma, h)\) has been appended to \( G \) in Step 2(a-iii) of \texttt{Infer}. In other words, \( Y \) is Step 2(a-iii) of \texttt{Infer}. Again, the algorithm must have proven some tuples prior to the execution of Step \( Y \), which contradicts the fact that \( J_P \) is the singleton set \( \{ t \} \) after the execution of Step \( Y \).

**Inductive hypothesis:** \( G \) is of length \( k \leq n \). Let \( X \) be any step during the execution of \texttt{ComputeOneRoute}_\( \mathcal{M} \)(\( I, J, J_s \)) such that \( G \) is of length \( k \leq n \). Then \( G \) is a route for the set \( J_P \) of all tuples which have been so far marked as proven by the algorithm.

Consider any step during the execution of \texttt{ComputeOneRoute}_\( \mathcal{M} \)(\( I, J, J_s \)) such that \( G \) is of length \( n + 1 \). Let \((\sigma, h)\) denote the last satisfaction step of \( G \) and let \( Y \) be the step in the execution of \texttt{ComputeOneRoute}_\( \mathcal{M} \)(\( I, J, J_s \)) when \((\sigma, h)\) has been appended to \( G \). Let \( t \) be the corresponding tuple which has been marked as proven in Step \( Y \). Let \( G' \) denote the sequence \( G \) without \((\sigma, h)\). In other words, \( G = G'.(\sigma, h) \), where \( G'.(\sigma, h) \) denotes the sequence obtained by appending \((\sigma, h)\) to \( G' \). Since \( G' \) has length \( n \), by the inductive hypothesis we have that immediately before Step \( Y \) is executed, \( G' \) is a route for \( J_P \setminus \{ t \} \). There are two possible cases to analyze, depending on the nature of Step \( Y \).

**Case 1:** \textit{Step \( Y \) is Step 2(a) of \texttt{ComputeOneRoute}.} It must be that \( \sigma \) is a s-t tgd which witnesses \( t \) with \( h \). Since \( G' \) is a route for \( J_P \setminus \{ t \} \) it immediately follows that \( G'.(\sigma, h) = G \) is a route for \( J_P \).

**Case 2:** \textit{Step \( Y \) is either Step 3(a-iii) of \texttt{ComputeOneRoute}, or Step 2(a-iii) of \texttt{Infer}.} This means that LHS\((h(\sigma))\) consists of only proven tuples. Hence, LHS\((h(\sigma)) \subseteq J_P \setminus \{ t \} \). Since
$G'$ is a route for $J_P \setminus \{t\}$, and $(\sigma, h)$ is appended to $G'$ in Step $Y$ to obtain $G$, it follows that $G'.(\sigma, h) = G$ is a route for $J_P$.

**Theorem 3.2.8** Let $\mathcal{M}$ be a schema mapping, $I$ be a source instance and $J$ be a solution for $I$ under $\mathcal{M}$. For every $J_s \subseteq J$, if there is a route for $J_s$, then the result of $\text{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)$ is a route for $J_s$.

**Proof.** Let $J_s$ denote the set of tuples $\{t_1, \ldots, t_n\}$. Since there exists a route for $J_s$, there must also exist a route for each $t_i$ in $J_s$. By the execution of $\text{ComputeOneRoute}$ on $J_s$, it is clear that every $t_i$ occurs in the set ACTIVETUPLES maintained by the algorithm. We shall prove the following statement: For any tuple $t$ in ACTIVETUPLES such that there exists a route for $t$, then $t$ is marked as proven by the time $\text{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)$ terminates. Then, it immediately follows by Proposition 3.2.3 that the result $G$ of $\text{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)$ is a route for $t$. In particular, $G$ is a route for every tuple $t_i$ in $J_s$, since $t_i$ occurs in ACTIVETUPLES, and there exists a route for $t_i$. Hence, $G$ is also a route for $J_s$, which completes the proof of the theorem.

We next prove the above statement. Let $t$ be a tuple which occurs in the set ACTIVETUPLES constructed during the execution of $\text{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)$. We show that if there exists a route for $t$, then $t$ is marked as proven by the time $\text{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)$ terminates. Let $R$ be a route for $t$. Without loss of generality, we shall assume that $R$ is a minimal route for $t$. The proof of the statement is by induction on the rank of $t$ in $R$.

**Base case:** $\text{rank}(t) = 1$. Let $t$ be a tuple which occurs in ACTIVETUPLES such that there exists a minimal route $R$ for $t$ and $\text{rank}(t) = 1$ in $R$. This means that $R$ must contain a single
satisfaction step \((\sigma, h)\), where \(\sigma\) is a s-t tgd. Hence, it must be that the invocation of findHom in Step 2 of \texttt{ComputeOneRoute} finds a satisfaction step \((\sigma', h')\) to witness \(t\), where \(\sigma'\) is a s-t tgd. Hence, \(t\) is marked as proven in Step 2(a) of \texttt{ComputeOneRoute}.

\textit{Inductive hypothesis:} \(\text{rank}(t) \leq n\). Let \(t\) be a tuple which occurs in \texttt{ACTIVETUPLES} such that there exists a minimal route \(R\) for \(t\) and \(\text{rank}(t) \leq n\) in \(R\). Then \(t\) is marked as proven by the time \texttt{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)\) terminates.

Let \(t\) be a tuple which occurs in \texttt{ACTIVETUPLES} such that there exists a minimal route \(R\) for \(t\) and \(\text{rank}(t) = n + 1\) in \(R\). We show that \(t\) is marked as proven by the time \texttt{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)\) terminates.

Let \((\sigma, h)\) denote the last satisfaction step in \(R\). Since \(R\) is a minimal route for \(t\), it must be that \(t\) is witnessed in the last satisfaction step of \(R\) with a target tgd, since \(\text{rank}(t) > 1\). Let \(\{t_1, \ldots, t_k\}\) be the tuples in \texttt{LHS}(\(h(\sigma)\)). Denote as \(R'\) the route obtained from \(R\) by removing its last satisfaction step. Clearly, \(R'\) is a minimal route for \texttt{LHS}(\(h(\sigma)\)) of rank \(n\). Hence, it must be possible to extract routes \(R_1, \ldots, R_k\) from \(R'\) such that \(R_i\) is a minimal route for \(t_i\), \(1 \leq i \leq k\). Since \(\text{rank}(R') = n\), it follows that \(\text{rank}(t_i) \leq n\) in \(R_i\). (A similar argument has been made in the proof of Lemma 3.2.1.)

Consider the step in the execution of \texttt{ComputeOneRoute}_{\mathcal{M}}(I, J, J_s)\) when \(t\) was added to \texttt{ACTIVETUPLES}. The procedure findHom is invoked with \(t\) in Step 2 of the algorithm. If findHom finds a pair \((\sigma', h')\) to witness \(t\), where \(\sigma'\) is a s-t tgd, then \(t\) is immediately marked as proven in Step 2(a) and we’re done. Otherwise, the algorithm proceeds to Step 3. Let \((\sigma', h')\) denote a pair which is considered in Step 3 of the algorithm. There are two cases to analyze, depending on whether \texttt{LHS}(\(h'(\sigma')\)) consists of only proven tuples or not (i.e., corre-
sponding to the If, and respectively, Else branches of Step 3(a)).

Case 1: LHS($h'(\sigma')$) consists of only proven tuples. Hence, $t$ is immediately marked as proven in Step 3(a-i) of the algorithm and we’re done.

Case 2: LHS($h'(\sigma')$) does not consist of only proven tuples. Hence, the triple $(t, \sigma', h')$ is added to UNPROVEN (Step 3(a-iii)) and FindRoute(LHS($h_m(\sigma_m)$)) is invoked next (Step 3(a-iv)).

There are two cases to consider here, depending on the status of $t$ after the recursive call to FindRoute(LHS($h'(\sigma')$)) returns and ComputeOneRoute proceeds to Step 3(a-v).

Case 2(a): The status of $t$ is proven. This is the easy case in which we’re immediately done.

Case 2(b): The status of $t$ is not proven. This means that the triple $(t, \sigma', h')$ is still UNPROVEN at Step 3(a-v) of the algorithm. In this case ComputeOneRoute returns to Step 3 and tries to find another way of witnessing $t$ with a target tgd. It is easy to see that in Step 3, ComputeOneRoute exhaustively tries all possible pairs for witnessing $t$, until $t$ is successfully proven in Steps3(a-i), or Step 3(a-v) of the algorithm. In the worst case, $t$ is still unproven after all possible pairs have been considered. Since $(\sigma, h)$ is one such pair which witnesses $t$, it follows that $(\sigma, h)$ must have been considered as well.

Consider Step 3 of ComputeOneRoute when $(\sigma, h)$ is found by findHom. It must be that not all tuples in LHS($h(\sigma)$) are proven. Otherwise, ComputeOneRoute would have obviously marked $t$ as proven in Step 3(a-i), which contradicts our assumption that $t$ is still unproven. Hence, ComputeOneRoute must have continued with the Else branch of Step 3(a).

It follows that the triple $(t, \sigma, h)$ must have been added to UNPROVEN is Step 3(a-iii) and FindRoute(LHS($h(\sigma)$)) must have been invoked in Step 3(a-iv). Hence, we have that for each tuple $t_i$ among $\{t_1, \ldots, t_k\}$ ($1 \leq i \leq k$) in LHS($h(\sigma)$), $t_i$ is in ACTIVETUPLES. Since for
each \( t_i \), there exists a minimal route \( R_i \) with \( \text{rank}(t_i) \leq n \) in \( R_i \) and \( t_i \) is in \text{ACTIVETUPLES}, it follows by the inductive hypothesis that \( t_i \) is marked as proven by the time the algorithm terminates. Consider the step of the execution when the last of the tuples in \( \{t_1, ..., t_k\} \), call it \( t' \), is marked as proven. We show that \( t \) is inferred as proven. There are three cases to analyze, depending on the step of the algorithm where \( t' \) has been marked proven.

- The tuple \( t' \) has been marked as proven in Step 2(a) of \text{ComputeOneRoute}. Hence, \text{Infer}\{t'\} is invoked in Step 2(b) and \( t \) is marked as proven in Step 2(a-iii) of \text{Infer}, since \((t, \sigma, h)\) is in \text{UNPROVEN} and all tuples in \text{LHS}(h(\sigma)) are proven.

- The tuple \( t' \) has been marked as proven in Step 3(a-i) of \text{ComputeOneRoute}. Hence, \text{Infer}\{t'\} is invoked in Step 3(a-ii) and \( t \) is marked as proven in Step 2(a-iii) of \text{Infer}, by the same argument as in the previous case.

- The tuple \( t' \) has been marked as proven in Step 2(a-iii) of \text{Infer}. Hence, \( t' \) must have been added to the data structure \( S \) in Step 2(a-i) in same iteration of the Repeat-Loop of \text{Infer}. Observe that the Repeat-Loop of \text{Infer} is exhaustively repeated until \( S \) is empty. Since \( S \) contains at least one tuple (i.e., \( t' \)), it must be that \text{Infer} continues with the next iteration of the Repeat-Loop, and it must be that \( t \) is marked as proven in Step 2(a-iii) in this iteration, since \((t, \sigma, h)\) is in \text{UNPROVEN} and all tuples in \text{LHS}(h(\sigma)) are proven.
3.3 Extending to the Nested Relational Model

Section 3.2 describes algorithms for computing routes for selected target data in the context of schema mappings where both the source and target schemas are relational. This section discusses an extension of these algorithms to handle schema mappings where the source and target schemas are hierarchical. A major difference which arises in this context is the notion of logical tuples. Intuitively, a logical tuple of a target data represents all the values found along the path from the root of the target nested instance to the target data selected by the user.

The next section describes the nested relational (NR) model and nested schema mappings [77, 100] which are used in Clio for modeling hierarchical (or XML) schemas and instances, as well as specifying the relationships between elements of such schemas. All the results described in Section 3.2 (Propositions 3.2.1 and 3.2.2 and Theorems 3.2.5 and 3.2.8) carry over to the NR model.

3.3.1 The Nested Relational Model

In the nested relational model described in [77], a nested relational schema is a set of labels \{R_1, ..., R_k\}, called roots, where each root is associated with a type \(\tau\), defined by the following grammar:

\[
\tau ::= \text{String} \mid \text{Int} \mid \text{SetOf} \ \tau \mid \text{Rcd}[l_1: \tau_1, ..., l_n: \tau_n] \mid \text{Choice}[l_1: \tau_1, ..., l_n: \tau_n].
\]

The types String and Int are atomic types. The types Rcd and Choice are complex types. A (record) value of type \(\text{Rcd}[l_1: \tau_1, ..., l_n: \tau_n]\) is a set of label-value pairs \([l_1: a_1, ..., l_n: a_n]\), where \(a_1, ..., a_n\) are of types \(\tau_1, ..., \tau_n\), respectively. A value of type \(\text{Choice}[l_1: \tau_1, ..., l_n: \tau_n]\)
\( \tau_n \) is a single label-value pair \( [l_k : a_k] \), where \( a_k \) is of type \( \tau_k \). The set type \( \text{SetOf} \ \tau \) (where \( \tau \) is a complex type) is used to model repeatable elements. Order is not considered, hence \( \text{SetOf} \) represents unordered sets. A value of type \( \text{SetOf} \ \tau \) is represented by a \( \text{SetID} \) and an associated (possibly empty) set of “children” values \( \{v_1, \ldots, v_m\} \), where each \( v_i \) is of type \( \tau \).

For simplicity, this section assumes that XML schemas are modeled using a single schema root of type record, where the type of each label is a set of records. The implementation of SPIDER, however, handles the nested relational model in its full generality.

**Example 3.3.1**  Figure 3.7 shows a schema mapping where both the source and target schemas are nested. The type information for atomic elements is omitted from the figure, to avoid clutter. Leaf elements have atomic types and record types have subelements represented by their indentations. The source and target schemas are similar to the source and target schemas of Figure 3.1, except the information is now organized hierarchically. The source schema \( S \) consists of a root label \( \text{Manhattan-Credit} \) associated with a record type. The record type has one label \( \text{Cards} \), which is a set of records. Every label within the record type of \( \text{Cards} \) is associated with an atomic type, except the label \( \text{SuppCards} \), which is a set of records. The target schema \( T \) consists of a root label \( \text{Fargo-Finance} \) which is associated with a record type with two labels, \( \text{Clients} \) and \( \text{Accounts} \). Since a client may have more than one account, the \( \text{Client} \) record contains a label \( \text{AccOwned} \) which is a set of records containing account numbers. The association between the clients and their accounts is made by means of the foreign key \( m_3 \) in Figure 3.7. The constraint \( m_3 \) is explained in more detail shortly.

The top of Figure 3.8 shows an instance \( I \) for the source schema \( \text{Manhattan-Credit} \).
Figure 3.7: A XML-to-XML schema mapping

The root label Manhattan-Credit is a record. The label Cards in this record is a set of two records about J. Long. In the first record, the SuppCards value is a SetID $Sc_1$ associated with two records about A. Long and M. Bent, respectively. The SuppCards value in the second record of Cards is a SetID $Sc_2$ and there are no records associated with it. The SetID $Sc_2$ is omitted from Figure 3.8.

Nested Schema Mappings For conciseness, we shall describe only a fragment of the nested schema mapping language of [100] in this section. The implementation of SPIDER, however, adopts the full mapping language described in [100] (which subsumes the nested tgd and nested egd language described shortly), and can handle all types described in the nested relational model. A nested tgd has the form $\forall x(S_1(x_1) \wedge \ldots \wedge S_m(x_m)) \rightarrow \exists y(T_1(x, y_1) \wedge \ldots \wedge T_n(x, y_n))$. 

140
Figure 3.8: Source instance $I$ and target instance $J$ satisfying the schema mapping of Figure 3.7.

The left-hand-side is a sequence of $S_i$ atoms, where $S_1$ is a path $r.s$ such that $r$ is a root label and $s$ is a label of the record $r$ whose associated type is a set of records. (Recall that for simplicity, an XML schema is assumed to consist of a single root of type record with components of set of records type.) In addition, every $S_i$, where $2 \leq i \leq m$, is either a path similar to $S_1$ or a SetID variable, and $x = x_1 \cup \ldots \cup x_m$ and $y = y_1 \cup \ldots \cup y_n$. Also, if $S_i$, where $2 \leq i \leq m$, is a SetID variable, then $S_i$ must occur among $x_1, \ldots, x_{i-1}$. A similar rule applies to the right-hand-side of the nested tgd with the sequence of $T_j$ atoms. In addition, every SetID variable that occurs on the right-hand-side must be existentially-quantified. In other words, a SetID cannot be “copied” by the tgd. A nested egd is of the form $\forall x (S_1(x_1) \land \ldots \land S_m(x_m)) \rightarrow x_1 = x_2$, where the sequence of $S_i$ atoms follow a similar rule as described earlier, and $x_1$ and $x_2$ are non-SetID variables that occur among $x$.

**Example 3.3.2** Figure 3.7 shows two source-to-target nested tgds, $m_1$ and $m_2$, a nested target tgd $m_3$, and a nested target egd $m_4$. The universally-quantified variables are omitted for conciseness. The nested tgd $m_1$ specifies how Cards facts in the source schema relate to
Clients and Accounts facts in the target schema. Note that the SetID variable Ao is existentially quantified in \( m_1 \). Similarly, \( m_2 \) specifies that for each supplementary cardholder in the source instance, there must exist a client in the target instance with the corresponding values. The variable Sc is a SetID variable and \((s', n', a)\) binds to a record of SuppCards. The target tgd \( m_3 \) requires that if a client is associated with an account, then the account information appears as a fact in Accounts. Finally, \( m_4 \) states that ssn is a key for Clients.

An instance \( J \) of the target schema Fargo-Finance is shown at the bottom of Figure 3.8. Together with the source instance \( I \) (also shown in Figure 3.8), this target instance satisfies the schema mapping shown in Figure 3.7. Conceptually, the target instance \( J \) can be obtained by applying a nested extension of the chase [76, 100] on the instance \( I \). Schema mapping systems such as Clio or RDA, however, use queries to generate \( J \). The nested chase procedure is similar in spirit to the original chase procedure used in the context of relational-to-relational data exchange [44]. For each combination of facts in the source that match the left-hand-side of a nested tgd, corresponding facts are added in the target so that the right-hand-side of the tgd is satisfied. For example, the source record \((434, \text{J. Long, Smith, 50K, Seattle, 6689, 15K, } Sc_1)\) in Cards matches the left-hand-side of \( m_1 \). As a consequence, the record \((434, \text{J. Long, Smith, 50K, Seattle, } Ao_1)\) is added to Clients, the record \((6689)\) is added to \( Ao_1 \) and the record \((6689, 15K)\) is added to Accounts in the target. These records are illustrated as \( t_1, t_4 \) and respectively, \( t_8 \) in the target instance \( J' \) in Figure 3.8. Here, \( Ao_1 \) is a newly generated SetID. Similarly, more tuples are added during the nested chase procedure to arrive target instance \( J \) illustrated in Figure 3.8.
3.3.2 Computing Routes in the Nested Relational Model

This section describes how the route algorithms presented in Section 3.2 can be extended to handle schema mappings where the source and target schemas may be hierarchical.

Facts and Logical Tuples In a relational target instance, a user selects one or more relational tuples from the target instance for which she wishes to compute routes. With a hierarchical target instance, a user selects one or more facts from the target. A fact $S(a)$ in an instance $J$ is a record, where $S$ is either a SetID, or a path $r.s$ such that $r$ is a root label and $s$ a label of the record $r$ whose associated type is a set of records. In the latter case, where $S$ is a path $r.s$, $S(a)$ is called a root fact. For example, $Ao_1(6689)$ is a fact, while Fargo-Finance.Accounts(1222,5K) is a root fact and they are records of the sets AccOwned and Accounts, respectively. In a hierarchical instance, a fact exists only in the context of its ancestor facts, if any. For example, there is only one ancestor fact for $Ao_1(6689)$ and it is Fargo-Finance.Clients(434, J.Long, Smith, 50K, A_1, Ao_1). There are no ancestor facts for Fargo-Finance.Accounts(1222,5K) because it is a root fact (i.e., this record is the first record that occurs in a set along the path from the root to this record). It is easy to see that every fact $S(a)$ induces a unique maximal sequence of ancestor facts, starting with a root fact. The logical tuple associated with the fact $S(a)$ in an instance $J$ is either the sequence $[S(a)]$, if $S(a)$ is a root fact, or, the sequence $[S_1(a_1), ..., S_k(a_k), S(a)]$, where $[S_1(a_1), ..., S_k(a_k)]$ is the sequence of ancestors facts of $S(a)$ in $J$ such that $S_1(a_1)$ is a root fact and every $S_i$, where $2 \leq i \leq k$, is a SetID such that $S_i$ occurs among $a_{i-1}$ and $S$ occurs among $a_k$. For example, the logical tuple associated with the fact $Ao_1(6689)$ in the target instance $J$ shown in Figure 3.8 is [Fargo-Finance.Clients(434, J.Long, Smith, 50K, Seattle, 143
In SPIDER, a user selects a set of facts in $J$ and SPIDER will compute one or all routes for the set of logical tuples associated with the set of selected facts. A logical tuple of a fact is needed to uniquely identify the fact. To see this, it is important to note that a user does not see the SetIDs in the XML document that represents the instance $J$ in Figure 3.8. For example, the target schema would require that every record in AccOwned be created under an element name AccOwned (and not a SetID) in the XML representation. Consider a selected fact $Ao_1(6689)$. Even though the notation $Ao_1(6689)$ unambiguously states that this is the AccOwned record of J. Long, it is important to note that a user does not see the SetID $Ao_1$. Since there may be more than one AccOwned record with the same account number 6689 that exist under different clients, a logical tuple of the selected record is therefore needed to identify the correct AccOwned record.

Similar to Definition 3.2.2, a route for a set of logical tuples $J_s$ of a solution is a finite non-empty sequence of satisfaction steps of the form $(I, \emptyset) \xrightarrow{m_1,h_1} (I, J_1) \ldots (I, J_{n-1}) \xrightarrow{m_{n-1},h_{n-1}} (I, J_n)$ which satisfies the conditions in Definition 3.2.2, where a set of logical tuples $K_1$ is contained in a nested instance $K_2$ if every logical tuple in $K_1$ occurs as a logical tuple in $K_2$. For example, $((I, \emptyset), (I, J_1), m_1, h)$ is a route for the logical tuple $[t_1, t_5]$, where $J_1 = \{t_1, t_5, t_9\}$ and $h=\{s \mapsto 434, n \mapsto J. Long, m \mapsto Smith, sal \mapsto 50K, A \mapsto A_1, Ao \mapsto Ao_1, cn \mapsto 1222, l \mapsto 5K, loc \mapsto Seattle, Sc \mapsto Sc_2\}$. This route states that, through $m_1$, the second tuple in the Cards relation leads to the creation of $t_1$, $t_5$ and $t_9$.

The algorithms for computing one and all routes in the context of the NR model are essentially as described in Section 3.2. The main difference consists in the findHom procedure.
(Figure 3.3), which is now modified to handle logical tuples, as opposed to relational tuples. This modification is illustrated with an example next.

**Example 3.3.3** Consider an invocation of \texttt{findHom} with arguments \((I, J, L, m_2)\), where \(I, J,\) and \(m_2\) are from Figures 3.7 and 3.8, and \(L\) is the logical tuple \(L = [t_2, t_6]\) associated with the fact \(t_6\) in \(J\). By “matching” \(L\) against the RHS of \(m_2\), Step 1 of \texttt{findHom} defines \(v_1\) as \(\{s' \mapsto 234, n' \mapsto A. Long, M \mapsto M_1, S \mapsto I_1, a \mapsto \text{California}, Ao \mapsto Ao_2, N \mapsto N_1\}\). Note that \(v_1\) contains assignments for \texttt{SetID} variables as well. When \(v_1\) is applied to the LHS of \(m_2\), the partially instantiated logical tuple \([\text{Manhattan-Credit.Cards}(s, n, m, \text{sal}, \text{loc}, \text{cn}, l, Sc), Sc(234, A. Long, \text{California})]\) is obtained. Hence, the assignment \(v_2\) (Step 2) is \(\{s \mapsto 434, n \mapsto J. Long, m \mapsto Smith, \text{sal} \mapsto 50K, \text{loc} \mapsto Seattle, \text{cn} \mapsto 6689, l \mapsto 15K, Sc \mapsto Sc_1\}\). With \(v_1 \cup v_2\), the LHS of \(m_2\) corresponds to the logical tuple \([\text{Manhattan-Credit.Cards}(434, J. Long, Smith, 50K, Seattle, 6689, 15K, Sc_1), Sc_1(234, A. Long, \text{California})]\), and the RHS of \(m_2\) is the logical tuple \(L = [t_2, t_6]\). Hence, the assignment \(v_3\) returned in Step 3 of \texttt{findHom} is empty. The algorithm then returns \(v_1 \cup v_2 \cup v_3\) (Step 4).

### 3.4 System Architecture

The implementation of \texttt{SPIDER} handles schema mappings where the source or target schemas are relational or hierarchical. The system is implemented in Java 1.5 on top of the Clio system. It uses the nested relational model as the underlying representation, and the mapping language of Clio [77, 100] to represent schema mappings. Currently, \texttt{SPIDER} operates with relational instances stored using DB2 UDB Personal Edition release 8.1.10, and XML instances.
stored as XML documents. DB2’s query engine and Saxon-SB 8.6 XSLT transformation engine are used to execute SQL and respectively, XSLT queries over relational and respectively, XML instances. A snapshot from SPIDER is illustrated in Figure 3.9.

In the findHom procedure (Figure 3.3), the required assignments \( v_1, v_2 \) and \( v_3 \) for a given tuple \( t \) are obtained as follows. First, \( v_1 \) is obtained by matching \( t \) against the RHS of the tgd \( \sigma \). Second, the LHS of \( \sigma \) is executed as a selection query (as indicated by \( v_1 \)) against the instance \( K \) to obtain all the assignments \( v_2 \) that agree with \( v_1 \). (Here, \( K \) is the source or target instance, depending on whether \( \sigma \) is a s-t or target tgd.) For each such \( v_2 \), all possible assignments \( v_3 \) that agree with \( v_2 \) are obtained by running the RHS of \( \sigma \) as an appropriate (based on \( v_2 \)) selection query on the target instance. Note that all possible assignments of \( v_2 \) and \( v_3 \) could in fact be obtained by running a single selection query (the join of the LHS and RHS of \( \sigma \)) against the source and target instances. While this may be more efficient for relational schema mappings, it was a design choice to run two separate queries, in order to handle general situations in which for example, the source instance is relational, while the target is XML. The assignments are fetched one at a time, as needed, from the result of the selection queries. For this reason, the implementation of ComputeOneRoute is scalable for relational instances. For XML instances, however, all the assignments are fetched at once, since the result produced by the Saxon engine is stored in memory.

### 3.4.1 Other Features of the Prototype

In addition to its ability to compute routes for selected target facts, SPIDER provides other features that we have implemented for the purpose of facilitating the process of under-
Figure 3.9: Snapshot from SPIDER: (a) source and target instances; (b) route forest; (c) detailed view of a satisfaction step in the route forest.

standing and debugging mappings. This section gives an overview of these features.

The visual interface of SPIDER consists of two main views: the schema mapping view, where a mapping designer is allowed to inspect the schema mapping, and the data view. Figure 3.9 shows a snapshot of the data view of SPIDER, which allows a mapping designer to explore the source and target instances (Figure 3.9(a)), and visualize routes (Figures 3.9(b–c)). Figure 3.9(b) shows SPIDER’s routes explorer, where a designer can examine a high-level
overview of all routes, or one route. At this level, the tuples and satisfaction steps involved are schematically depicted as rectangles and arrows, respectively. They are color coded, to allow a designer to quickly distinguish between source and target tuples, and, respectively, satisfaction steps with source-to-target or target tgds. The details of each satisfaction step can be visualized in a “watch” window (Figure 3.9(c)), as we shall describe shortly.

Computing Routes for Source Data Similar to computing routes for target facts, SPIDER can also compute routes for selected source facts. This feature is useful for identifying consequences of the selected source facts in the target. The algorithms for finding one or all routes for a set of source facts are similar in spirit to the algorithms for finding routes for selected target facts described in Section 3.2, except that every selected source fact \( t \) is now matched against the LHS clause of a s-t tgd that together with \( t \) (and possibly other source facts) can witness some target facts \( T \). The process is then repeated by trying to find a target tgd that, together with some of the facts in \( T \), witnesses some other target facts, and so on, until no more target facts can be witnessed.

Computing Alternative Routes SPIDER is not limited to the computation of a single route, and alternative routes can be computed on demand for a set of source or target facts. Alternative routes are computed by allowing \texttt{ComputeOneRoute} to continue searching for another route, as opposed to abandoning the computation as soon as one route is computed. If an alternative route is sought for some selected fact \( t \), the algorithm “undoes” the last choice of tgd and facts made when computing one route for \( t \) and proceeds with the next choice, if any.

Guided Routes Exploration At any step in a route, SPIDER is able to suggest several dependencies that can be applied, if any. The mapping designer may choose one of these candidates
and decide upon the direction of the route, thus “guiding” the exploration of the route as desired. At a later moment, the designer can reconsider her decision and explore some alternative segment of the route.

“Standard” Debugging Features Similar to debuggers for programming languages, SPIDER is equipped with standard features such as step-by-step routes computation, breakpoints and “watch” windows. Routes can be computed in one run, or step-by-step. In the latter mode, the designer is allowed to set breakpoints on specific dependencies that when reached, trigger a stop in the computation of the route (or the route forest). The designer can examine the subsequent steps one at a time, or she can decide to demand the continuation of the computation until reaching another breakpoint, or SPIDER concludes the computation of the route (or route forest). At each step, the designer may choose a specific tgd to be applied, thus guiding the
exploration of the route as explained above. Figure 3.10 illustrates a snapshot from SPIDER during the step-by-step computation of a route forest, where a breakpoint has been set on the dependency denoted as $D_2$ (highlighted). The dotted branches represent SPIDER’s suggestions of dependencies which may be further explored. The combination of breakpoints with guided routes computation is especially useful in computing all routes, in that it allows a designer to explore only parts of the route forest that she might be interested in (e.g., routes with $D_2$), without waiting for the entire route forest to be constructed.

When examining a step in a route, the mapping designer can “watch” the assignments for the facts in the tgd computed by the \texttt{findHom} procedure as illustrated in Figure 3.9(c). These facts are automatically highlighted on the source or target instances, as illustrated in Figure 3.9(a). In addition, SPIDER is equipped with a “watch” window where one can visualize the creation of the target instance while exploring routes.

**Schema-level Exploration of Schema Mappings** In addition to its data-driven debugging capabilities, SPIDER also facilitates the understanding of schema mappings directly at the level of the source and target schemas through the display of schema-level routes. Given a schema element $T_e$, SPIDER can schematically show the source or target schema elements that are directly or indirectly responsible for creating values for $T_e$, as well as other target schema elements whose values are copied from $T_e$. This feature allows a designer to focus on parts of the schema mapping that are relevant for selected source or target schema elements. Figure 3.11 shows a screenshot from SPIDER’s schema view, which displays schema-level routes for the \texttt{ACCHOLDER} attribute in the target. These routes indicate that account holder values are created from \texttt{SSN} and \texttt{CUSTSSN} values in the source (via dependencies denoted as $D_1$ and
$D_3$, respectively) and, furthermore, they contribute to the creation of target SSN (via a target dependency denoted as $C_1$).

### 3.5 Experimental Evaluation

This section describes the results of an experimental study conducted on both real and synthetic datasets to assess the efficiency of SPIDER in computing one or all routes, under the effect of various parameters. All experiments have been executed on a Pentium 4, 2.8GHz Windows machine with 2GB RAM. The DB2 buffer pool was set to 256MB. Each experiment was repeated three times and execution times averaged over the second and third runs are reported.
1 join
\[ S \leftarrow_{\text{suppkey}} L; O \rightarrow_{\text{custkey}} C; PS \leftarrow_{\text{partkey}} P; N \rightarrow_{\text{nationkey}} R \]

2 joins
\[ S \leftarrow_{\text{suppkey}} L \leftarrow_{\text{orderkey}} O; S \leftarrow_{\text{suppkey}} PS \leftarrow_{\text{partkey}} P; C \rightarrow_{\text{nationkey}} N \rightarrow_{\text{nationkey}} R \]

3 joins
\[ S \rightarrow_{\text{suppkey}} L \rightarrow_{\text{partkey,suppkey}} PS \rightarrow_{\text{partkey}} P; O \rightarrow_{\text{custkey}} C \rightarrow_{\text{nationkey}} N \rightarrow_{\text{nationkey}} R \]

Figure 3.12: Joins used in tgd in the relational and flat-hierarchy synthetic scenarios.

In summary, our experimental results show that both route algorithms can be executed at interactive speeds, under assumptions that are realistic in scenarios where understanding and debugging mappings is desired. Our experimental results also show that ComputeOneRoute can be efficiently executed, while ComputeAllRoutes may perform orders of magnitude slower compared to ComputeOneRoute, in general. This suggests that, although it is possible to compute all routes, the ability to compute one route fast, and exploit the “thinking time” to generate alternative routes in the background, is valuable in general. In some cases, the user may be satisfied with only one route, or a few alternative routes, which are faster to compute than all routes, in general. In certain scenarios, however, it is also important to be able to compute all routes, as the benefit provided by these routes may offset the inherent performance overhead in computing them. For example, if confidential information from the source has been leaked in the target database, the ability to compute all routes for this leaked target data is invaluable, as it allows a user to understand all aspects of the schema mapping \( \mathcal{M} \) that contribute to the leak, thus enabling her to correct \( \mathcal{M} \), and ensure that confidentiality is preserved when the data is migrated from the source to the target.
Figure 3.13: (a–c) Performance evaluation of ComputeOneRoute and (d) comparison in performance between ComputeOneRoute and ComputeAllRoutes in the relational synthetic scenarios.

3.5.1 The Synthetic Datasets

The three synthetic scenarios used in the experiments are called relational, flat-hierarchy and deep-hierarchy. The first two scenarios have been designed with the goal of measuring the influence of various parameters, when the source and target schemas are relational and respectively, hierarchical. The parameters are: the size of source (and target) instances, the number of tuples selected by a user, and the size and complexity of the schema mappings. The third scenario has been designed to measure the influence of the depth of the selected elements of an XML document on the performance of the route algorithms, where both
the source and target schemas are deeply nested.

**Relational Scenario** Four schema mappings $\mathcal{M}_0, \ldots, \mathcal{M}_3$ have been designed in this scenario. The subscripts denote the number of joins, which shall be explained shortly, used in the tgds of the schema mapping. In each schema mapping, the source schema conforms to the TPC Benchmark H (TPCH) Standard Specification Revision 2.1.0 [93] and consists of eight relations Customer (C), Lineitem (L), Nation (N), Orders (O), Part (P), Partsupp (PS), Region (R) and Supplier (S). The target schema consists of six “copies” of the source schema: for each relation $R_0$ in the source schema, there are six relations $R_i$, $i \in [1, 6]$, identical to $R_0$ in the target schema. Hence, the target schema can be viewed as having six groups of relations, where each group is a “copy” of the source schema. In the first schema mapping $\mathcal{M}_0$, the s-t tgds populate relations in the first group by copying every $R_0$ to the corresponding relation $R_1$ in the target. The target tgds are such that every relation $R_i$ in the $i$th group, $i \in [2, 6]$, is copied from the corresponding relation $R_{i-1}$. The second schema mapping $\mathcal{M}_1$ is similar except that every tgd in this schema mapping has 1 join on both sides corresponding to the 1 join case illustrated in Figure 3.12. For example, one such s-t “copying” tgd in $\mathcal{M}_1$ is $S_0(sk, ...) \land L_0(\ldots, sk, \ldots) \rightarrow S_1(sk, \ldots) \land L_1(\ldots, sk, \ldots)$, where the variable $sk$ corresponds to the supplier key attribute ($suppkey$) shown in Figure 3.12. The rest of the variables are omitted from the figure for conciseness. The schema mappings $\mathcal{M}_2$ and $\mathcal{M}_3$ are similar except that the tgds have 2 and, respectively, 3 joins on both sides, as shown in Figure 3.12. We remark that using only such “copying” tgds in these four schema mappings does not bias the empirical evaluation; SPIDER separately operates with the LHS and RHS of each tgd in $\text{findHom}$ and complexity of each side of the tgds has been varied in each schema mapping.
Figure 3.13(a) studies the influence of the size of the input (i.e., the size of source and target instances, as well as the number of tuples for which a route needs to be computed) on the performance of ComputeOneRoute for a fixed schema mapping. The sizes of \((I,J)\) are (10MB,60MB), (50MB,300MB), (100MB,500MB), (500MB,3GB) respectively. The number of selected tuples is varied between 1 and 20. To keep the comparison meaningful, all tuples have been selected at random from the same group of target relations so that the number of satisfaction steps in a route of each selected tuple, which we call the \(M/T\) factor, is kept constant. For example, \(M/T=2\) for tuples of relations in group 2, since these tuples are witnessed with one s-t tgd and one target tgd. Figure 3.13(a) illustrates the time required to compute one route for tuples in group 3 in the schema mapping with one join tgd (\(M_1\)). The running time increases as the number of selected tuples increases, since more reverse chase steps need to be taken, hence more queries are executed. As an example, only 6 queries are executed to find a route for one tuple, while as many as 140 queries need to be executed to find a route for 20 tuples. The execution time also increases with the size of the source and target instances. Routes for 10 and 20 tuples are computed in under 4, and respectively, under 8 seconds on the datasets with 10MB, 50MB and 100MB source instances. However, the performance degrades to a larger extent on the dataset where the source and target instances are of size 500MB and respectively, 3GB. This is explained by the fact that the queries, with joins involved, are now executed on larger instances. We note, however, that a user is unlikely to select as many as 20 tuples in the first place. We expect a user to be interested in a smaller number of tuples at any time, cases in which SPIDER still performs at interactive speeds.

Figure 3.13(b) analyzes the influence of the \(M/T\) factor on the performance of the
ComputeOneRoute algorithm, by computing routes for target tuples in different groups. We performed six runs, each time selecting up to 20 tuples from the same group (i.e., the M/T factor varies between 1 and 6). The source instance is fixed at 100MB in the schema mapping with 3 joins tgds ($M_3$). The running time increases with a higher M/T factor, since more intermediary tuples are discovered (and reverse chased) along the route to source tuples, hence more queries are executed. For example, it takes 1.8 seconds and, respectively, 2.9 seconds to find a route for a tuple with an M/T factor of 3 and respectively, 6. Only 10 queries are executed in the first case, while as many as 22 queries are executed in the latter case. Again, it can be observed that for a small number of selected tuples, the system performs well in general, requiring up to 3 seconds to find a route for three target tuples with an M/T factor of up to 5.

Figure 3.13(c) studies the influence of the complexity of schema mappings on the performance of the system. This time, we vary the schema mapping with 0 to 3 joins tgds, and we fix the M/T factor to 3 and the size of the source instance to 100MB. The running time of ComputeOneRoute increases with the number of joins in the tgds. This is explained by the fact that query execution degrades in performance with the number of joins. Still, the system performs well, taking up to 4.5 seconds to compute routes for a set of 7 target tuples, in all four schema mappings.

A similar suite of experiments has been conducted with ComputeAllRoutes, and similar trends have been observed. However, ComputeAllRoutes may perform much slower compared to ComputeOneRoute. Figure 3.13(d) shows a comparison between the running times of the two algorithms for the schema mapping with 1 join, the source instance of 100Mb and tuples with an M/T factor of 3. (Note the logarithmic scale). For 5 tuples, one route is found
and printed in 2 seconds, while ComputeAllRoutes requires about 100 seconds to construct the route forest. The running time shown for ComputeAllRoutes does not include the time required to print all routes from the route forest (algorithm NaivePrint). The performance gap between the two algorithms will be even larger if all routes are printed. We have found the gap between the execution times of the two algorithms somewhat surprising at first. However, our investigations revealed that the route forest constructed by ComputeAllRoutes can be quite large. For example, the route forest constructed for one selected tuple consisted of over 2000 satisfaction steps, whereas the route constructed by ComputeOneRoute for the same tuple consisted of only 3 satisfaction steps. This explains the performance overhead of ComputeAllRoutes, which can be orders of magnitude slower than ComputeOneRoute. As mentioned earlier, however, we expect smaller instances to be used for the purpose of understanding and debugging mappings, and our experiments with real datasets suggest that the performance of ComputeAllRoutes is acceptable in such cases.

Flat-hierarchy Scenario We have also studied the influence of the size of source (and target) instances, as well as the number of selected target elements for which a route needs to be computed, on the performance of ComputeOneRoute for the XML case. The source schema consists of a root record having eight sets of records nested underneath, each set corresponding to one TPCH relation. Similarly, the target schema consists of six “copies” of the source schema and the s-t and target tgds are similar to our relational scenario (i.e., they are “copying” tgds). Hence, this scenario deals only with elements nested immediately underneath the root (i.e., the depth is 1). The sizes of \((I, J)\) used in the experiments are \((500\text{KB},3\text{MB})\), \((1\text{MB},6\text{MB})\) and \((5\text{MB},30\text{MB})\) respectively. As expected, the running time of ComputeOneRoute increases
Varying the size of source and target instances
Nested TGDS with 1 join, Routes with $M/T=3$

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# logical tuples of length 1

I:500 KB; J:3 MB
I:1 MB; J:6 MB
I:5 MB; J:30 MB

Varying the depth of selected elements from 1 to 5
$I_{ST}=700$ KB, $J_{ST}=700$ KB

0.0 0.4 0.8 1.2 1.6
1 2 3 4 5 6 7 8 9 10 11 12
# selected elements

(a) (b)

Figure 3.14: Performance evaluation of ComputeOneRoute in the (a) flat hierarchy and (b) deep-hierarchy scenarios.

with the size of the source and target instances, as well as the number of selected target elements
that need to be justified, as shown in Figure 3.14(a). The system performs very well, requiring
at most 5 seconds to compute one route for 20 elements, for all three pairs of source and target
instances. The performance of the algorithm decreases with the increase of the $M/T$ factor, as in
the relational case. However, a more drastic decrease in performance has been noticed with the
increase in the number of joins in the tgds. This is explained by the fact that the free version of
the Saxon XSLT engine used in the reverse chase procedure does not perform join reordering,
and simply implements all for-each clauses as nested loops.

**Deep-hierarchy Scenario** To analyze the effect of the depth of selected elements on the per-
formance, we designed a schema mapping where the source and target schemas are identical,
and have the nesting Region/Nation/Customer/Orders/Lineitem. In other words, the root
consists in a set of regions, each region has nations nested underneath and so on. The set $\Sigma_{st}$
consists of one s-t tg'd that copies the source instance into an identical target instance, and there

158
are no target tgds. The performance of ComputeOneRoute has been tested on elements found at different nesting levels in the target instance. For example, the Customer and respectively, Lineitem elements have been selected for the experiments with levels 3 and 5, respectively. The results are shown in Figure 3.14. The execution time decreases with the depth of the selected element. Intuitively, with a deeper selected element, more variables will be instantiated in the selection queries generated by reverse chase. Hence, the resulting selection queries will execute faster. For elements of depth 1, the execution time for at most 5 selected region facts is reported in Figure 3.14, since there are only 5 distinct regions in the TPCH instance.

### 3.5.2 The Real Datasets

The route algorithms of SPIDER have also been evaluated using two real datasets (DBLP and Mondial) for which schema mappings have been designed in order to exchange bibliographical, and respectively, geographic information. Two DBLP data sources have been used in the DBLP scenario; one conforms to the original DBLP bibliography schema and one is from [35]. The first relational schema in the Amalgam integration test suite [69] has been used as target schema. The Mondial scenario uses the relational and nested versions of the Mondial
schema [68], as source, and respectively, target schemas. In both cases, the s-t tgds have been
generated with Clio and the foreign key constraints of the target schemas have been used as
target tgds. Some characteristics of the source and target schemas, the number of s-t and target
tgds, as well as the size of source and resulting target instances used in the experiments are
shown in Table 3.1. SPIDER was used to compute (one or all) routes for 1 to 10 randomly
selected target facts in both scenarios. Both route algorithms performed at interactive speeds
when at most 3 facts were selected. In these cases, the time required to find one route was under
0.1 seconds, while the route forest took at most 5 seconds to be constructed, in both scenar-
ios. On average, we have observed that `ComputeAllRoutes` performed around two orders of
magnitude slower compared to `ComputeOneRoute`. This is explained by the fact that more
queries are executed when the route forest is constructed. For example, 4 queries have been
executed in computing one route for a selected fact in the Mondial target database. In contrast,
as many as 250 queries were executed to construct the route forest for that same fact.

An interesting observation we made during this experiment is that the representation
output by our algorithm for computing all routes may embed many “similar” routes. Intuitively,
these similar routes consist of sequences of satisfaction steps which use the same tgds, but
different homomorphisms. In the Mondial scenario, for example, there may be many routes
that witness a `Country` fact with various different `City` facts, because of a target referential
constraint from `City` to `Country`. In some cases, a user may be satisfied with analyzing only
a subset of these routes, which differ “sufficiently” from one another. An interesting question
is whether our algorithm for computing all routes can be modified to output a representation
embedding only these routes efficiently (i.e., much faster than `ComputeAllRoutes`).
3.6 Related Work

A system for understanding, designing, and refining mappings by example is proposed in [99], in the context of mappings between relational schemas expressed as SQL queries. The Muse system [6, 7] extends [99] in that it operates with schema mappings, as opposed to SQL queries, and handles any combination of relational and hierarchical schemas. Furthermore, Muse provides additional functionality for illustrating and designing grouping semantics for schema mappings. These systems [6, 7, 99] study the problem of selecting good source and target instances that are illustrative for the behavior of the schema mappings. SPIDER is complementary to these systems, in that it allows a user to create and use any source instance that she thinks is representative for debugging. This is similar to using test cases for analyzing the correctness of a program during a software development cycle. It would be very desirable to incorporate the functionality of [6, 7, 99] in SPIDER. In particular, the instances generated in these systems can be naturally used as test cases in understanding and debugging schema mappings with SPIDER.

Commercial systems such as Altova MapForce [67] and DataDirect Stylus Studio [89] ship with integrated debugging facilities for data exchange. In specifying mappings, these systems rely directly on transformation languages such as SQL, or XQuery. Hence, their built-in debugging tools are SQL or XQuery debuggers. Furthermore, debugging is “source data-centric” in these systems, in that they provide functionalities for stepping through the transformation starting from selected source data only. In contrast, SPIDER operates at the higher level of abstraction offered by schema mappings, and can compute routes starting from selected
source data (Section 3.4.1), as well as selected target data.

Our route algorithms bear resemblance to top-down resolution techniques used in
deductive databases. Furthermore, the problem of computing a route is very related to the
problem of computing the provenance of data. The next two sections compare SPIDER with
related work in these areas in more detail.

3.6.1 Approaches in Deductive Databases

The route algorithms bear resemblance to top-down approaches such as the SLD res-
such as query/subquery (QSQ) [97], rule/goal graphs [95] or OLDT [90] are even more closely
related. These approaches use memoization to avoid redundant computations and infinite loops.
Memoization is an optimization technique that avoids repeated computations by storing the re-
sults of internal function calls. Whenever a function is called with previously processed inputs,
the result is retrieved from the memory, rather than recomputed. Our route algorithms also avoid
redundant computations and infinite loops by not exploring any branches under repeated tuples.
Explored branches are never discarded and they are memoized. In fact, our algorithm for com-
puting all routes can be seen as an extension of such sophisticated top-down datalog evaluation
techniques in the context of schema mappings. A difference between our route algorithms and
these top-down approaches is that we make use of the target instance, which is available to us.
In contrast, the result of a datalog program is not available during resolution.

The following simple example illustrates the difference. Consider the schema map-
ning $\mathcal{M}$, where $\Sigma_{st}$ consists of two s-t tgds: $\sigma_1 : S_1(x, y) \rightarrow T_1(x, y)$ and $\sigma_2 : S_2(x, y, u) \rightarrow$
\( T_2(x, y, u) \), and \( \Sigma_t \) consists of a target tgd \( \sigma_3 : T_1(x, y) \land T_2(y, z, u) \rightarrow T_3(x, z) \). Observe that these tgds can be interpreted as a datalog program, \( P \). Consider the target instance \( J = \{ T_1(1, 2), T_3(1, 3) \} \) which is a solution for the source instance \( I = \{ S_1(1, 2) \} \) under \( \mathcal{M} \). (Note that there is no route for \( T_3(1, 3) \).) In determining whether the result of the datalog program \( P \) on \( I \) contains the tuple \( T_3(1, 3) \), a top-down approach (e.g., [90, 95, 97]) would essentially proceed as follows. First, \( \sigma_3 \) is considered. The subgoal \( T_3(x, z) \) is unified with the tuple \( T_3(1, 3) \) and the computation continues with evaluating \( T_1(1, y) \land T_2(y, 3, u) \). At this point, the partially instantiated subgoal \( T_1(1, y) \) is chosen for expansion and unified with \( T_1(x, y) \) in the right-hand-side of \( \sigma_1 \). The computation continues with \( S_1(1, y) \) whose evaluation on the source instance results in the assignment \( y \mapsto 2 \). This assignment is passed to the remaining subgoal \( T_2(y, 3, u) \) and the computation proceeds with the evaluation of \( T_2(2, 3, u) \), which leads to \( S_2(2, 3, u) \) (after the unification with the right-hand-side of \( \sigma_2 \)). Since there are no source facts of the form \( S_2(2, 3, u) \), the evaluation concludes at this point with the answer “no”. In contrast, the route algorithms leverage the solution \( J \). Hence, they avoid redundant computations earlier. Each reverse chase step with a target tgd obtains completely instantiated facts from the target instance. For the example, the reverse chase step with \( T_3(1, 3) \) does not return any assignments. Thus, it can be immediately concluded that there is no route for \( T_3(1, 3) \). On the other hand, the top-down techniques will continue the evaluation up to the source tuples. It is only when the source predicate \( S_2(2, 3, u) \) cannot be successfully instantiated that they will conclude \( T_3(1, 3) \) is not in the result. An advantage of using the target instance \( J \) is that the computation can be pushed to the database by using queries in each reverse chase step, as described in Section 3.4. Furthermore, the polynomial running time of the route algorithms
(Propositions 3.2.1 and 3.2.2) follows immediately. To the best of our knowledge, a similar result has not been given in the context of top-down datalog evaluation.

One may also argue that we could have used the magic set rewriting technique [15] to obtain all the tuples that contribute to the creation of the selected target tuples. However, one must first develop an extension of this technique to handle schema mappings. Furthermore, it remains an interesting open question whether the magic set technique can be used to generate a compact, polynomial-size, representation of all routes.

### 3.6.2 Computing the Provenance of Data

Figure 3.15 shows a classification of systems for tracing data provenance, reproduced from Figure 1.3 for convenience. In contrast to DBNotes, which adopts the eager approach for computing provenance over SQL queries, SPIDER adopts a lazy approach in computing routes for selected target data: routes are computed only when needed, by examining the source instance, the target instance and the schema mapping. It was a design choice not to adopt the eager approach in computing routes, since this approach may involve reengineering the underlying system. By adopting a lazy approach, SPIDER can readily work on top of Clio [56],
RDA [81], or other data exchange and data integration systems that are based on a similar formalism for schema mappings.

The rest of this section describes a comparison between SPIDER and eager approaches for computing provenance over schema mappings that have been implemented in [51, 53, 96]. It also describes the differences between SPIDER’s route algorithms and the lazy approach of Cui et al. [37, 38] for tracing data provenance over SQL queries. A comparison between eager and lazy approaches that have been developed for computing provenance over SQL queries has been described in Section 2.7.

The MXQL system of [96], and recently ORCHESTRA [51, 53], adopt the eager approach for computing provenance over schema mappings. In these systems, extra information from the execution is kept so that provenance can be easily answered later. In [96], the authors propose a concept of provenance at the level of schema mappings for data exchange. The underlying data exchange transformation engine is reengineered so that additional information about which source schema elements and mappings contributed to the creation of a target data is propagated along and stored with the target data. In particular, it modifies the way queries are generated in Clio in order to capture additional information during the exchange. This information can later be queried using a special query language called MXQL. SPIDER is similar to [96] in that it operates over relational or XML schema mappings. However, SPIDER is different from [96] in two aspects. First, SPIDER can be used “as is” on information integration systems based on similar formalisms for schema mappings. It does not require changes to the underlying engine. Second, routes can be automatically computed for any source or target data selected by a user and these routes contain information about schema-level, as well as data-level
provenance. In contrast, the approach of [96] requires a user to be familiar with MXQL to query about schema-level provenance, while data-level provenance is not considered.

**ORCHESTRA [51, 53]** is a peer-to-peer collaborative data sharing system where provenance of data is automatically propagated along and stored with the data, as data is being transformed from one peer to another based on schema mappings. In contrast to the MXQL system, the provenance information stored in ORCHESTRA contains details about both schema-level and data-level provenance. This information is leveraged for filtering updates based on trust conditions, as well as incrementally maintaining the peer instances when updates occur in the system.

Another related system which adopts the eager approach for computing provenance is Explain [11], an explanative module for the CORAL deductive database system. Explain records additional information during the execution of a rule-based program and uses this information for explaining how a certain conclusion is reached, as well as identifying consequences of a certain fact produced by the program.

The lazy approach of Cui et al. [38] for computing provenance of data over relational views expressed as SQL queries has been overviewed in Section 2.7. Briefly, the provenance of a tuple in a view is described as the tuples in the base tables that witness the existence of that tuple. Whenever the provenance of a view tuple \( t \) is sought, the approach of [38] is to generate a query to retrieve all combinations of base tuples that together with the query justify the existence of \( t \). There are several differences between the route algorithms of SPIDER and the approach of [38]. First, the pair \((I, J)\) is part of the input to the route algorithms, where \( J \) is a *solution* for \( I \) under the schema mapping. Since \( J \) is any solution, there may exist tuples in \( J \) with *no*
routes. In contrast, in the context of [38], the equivalent of $J$ is the output of an SQL query executed over $I$. Consequently, the provenance of every tuple in $J$ always exists. Second, the algorithms for computing routes work in the context of schema mappings where the source and target schemas can be nested, while the approach of [38] handles only relational views defined over relational sources. Third, [38] handles aggregates and negation. The language of schema mappings we consider cannot express aggregates or negation. The fourth difference lies in the representation of provenance. The route algorithms of SPIDER operate with schema mappings, and not with SQL queries. Hence, a tuple in $J$ may relate to several other intermediate tuples in $I$ and $J$ through possibly different tgds. These “intermediate relationships” between tuples are captured in routes. In contrast, the provenance of a tuple $t$ as defined in [38] is the set of source tuples that witness $t$ according to the SQL query. An example that exemplifies this difference is discussed shortly. Finally, the language of schema mappings allows one to define recursive computations. In contrast, recursive views are not handled in [38]. Even if recursive views were handled in [38], the description of provenance only with source tuples can be unsatisfactory as the following example illustrates.

Consider a schema mapping $\mathcal{M}$ where $\Sigma_{st}$ consists of one s-t tgd $\sigma_1 : S(x, y) \rightarrow T(x, y)$ and $\Sigma_t$ consists of a target tgd $\sigma_2 : T(x, y) \land T(y, z) \rightarrow T(x, z)$ that defines the transitive closure of the binary relation $T$. Consider a source instance $I = \{s_1 : S(1, 2), s_2 : S(2, 3)\}$ and the target instance $J = \{t_1 : T(1, 2), t_2 : T(2, 3), t_3 : T(1, 3)\}$ which is a solution for $I$ under $\mathcal{M}$. Clearly, the two source tuples witness $t_3$ with $\mathcal{M}$, but this is not as informative as showing a route $s_1 \xrightarrow{\sigma_1} t_1, s_2 \xrightarrow{\sigma_1} t_2, \{t_1, t_2\} \xrightarrow{\sigma_2} t_3$ that describes all the intermediate relationships. In particular, the fact that $t_3$ is a consequence of $t_1$ and $t_2$ with $\sigma_2$ is captured
in the route. (For simplicity, we have omitted the homomorphisms in each step and showed only the relevant tuples in the source instance.) The process of computing a route for \( t \) is more complex in SPIDER, since it is no longer sufficient to ask a query over the source instance as in [38].

As explained in Section 2.7, the type of provenance studied by Cui et al. [37, 38] is called *why-provenance* [27]. As the name suggests, why-provenance is concerned with the combination of source tuples which explain why a tuple \( t \) is in the result of a query. Our notion of route differs from why-provenance in that it explains, in addition, how source tuples have been combined to produce \( t \). In this sense, routes are a form of *how-provenance* in the terminology of [52]. How-provenance is more general than why-provenance, in that the later can be derived from the former, whereas the converse is not true. The type of provenance considered in the ORCHESTRA system [51, 53], which we described earlier, is also a form of how-provenance. However, ORCHESTRA uses the eager approach, whereas to date, SPIDER is the only system that adopts the lazy approach for computing provenance over schema mappings.
Chapter 4

The Schema Integration System

So far, we have described two approaches for enabling data interoperability that are based on tracing data provenance through mappings. In this chapter, we turn our attention to schema integration, which is another important aspect in providing support for data interoperability. Schema integration is the problem of creating a unified target schema from a set of existing source schemas that relate to each other, possibly via correspondences between their elements, or via some other forms of schema mappings such as constraints or views. By providing a standard representation of the data, the integrated target schema can be viewed as a means for dealing with heterogeneous data sources.

The schema integration problem is encountered in data integration and in several other related contexts. In data integration [65], the unified schema yields a single access point against which queries are posed to access a set of heterogeneous sources. Other applications include consolidating data sources of merged organizations into one database or warehouse, and integrating related application silos to create aggregated intelligence. In general, schema integration
is a form of “metadata chaos” reduction: quite often, many overlapping schemas (variations or
evolutions of each other) exist, even in the same computer, and need to be consolidated into
one. Schema integration is recognized as one of the building blocks for metadata applications
in the model management framework of Bernstein [19].

In this chapter, we describe a method and a design system that provide: 1) adaptive
enumeration of multiple interesting integrated schemas, and 2) easy-to-use capabilities for refin-
ing the enumerated schemas via user interaction. The method operates at a logical, conceptual
level that abstracts away the physical details of relational or XML schemas and makes it easy to
express user requirements. Furthermore, our method is a departure from previous approaches
to schema integration, which focus on the generation of a single integrated schema, and do not
offer a systematic exploration of the possible integrated schemas.

We assume that we are given a set of two or more source schemas (describing data
in a common domain) together with a set of correspondences that relate pairs of elements in
these schemas. Correspondences signify “semantically equivalent” elements in two schemas.
They can be user-specified or discovered through schema matching techniques [80]. Given
such input (source schemas and correspondences), our enumeration approach generates many
possible meaningful design choices for the integrated target schema. At the same time, we also
generate, for each choice of an integrated schema, the schema mapping that specifies how the
data in each of the source schemas is to be transformed to the integrated schema. Once the
mappings are obtained, test source data may be exchanged under the integrated schema, and
our SPIDER system, described in Chapter 3, can be immediately applied for the purpose of
facilitating a user’s understanding of the generated schema mappings.
Figure 4.1: Overview of our schema integration methodology.

A high-level overview of our schema integration methodology is schematically illustrated in Figure 4.1, for the case of two source schemas. We note that we are able to handle any number of source schemas, in general. The architecture is the same except that we require correspondences between multiple pairs of schemas.

As an initial step in our approach (Step 1 in Figure 4.1), each source schema, with its constraints and nesting, is recast into a higher-level graph of concepts with `HasA` relationships. Each concept is essentially a relation name with an associated set of attributes. A concept intuitively represents one category of data (an entity type) that can exist according to a schema (e.g., “department”, “employee”, etc.). Concepts in a schema may have references to other concepts in the schema and these references are captured by `HasA` edges (e.g., “employee” contains a `HasA` reference to “department”). For the most part, our subsequent integration method operates at the higher level of abstraction that is offered by the concept graphs.

Next, we identify matching concepts in different graphs by taking into account correspondences between their attributes. For every pair of matching concepts we then have the
alternative of merging them into one integrated concept or of leaving them as separate concepts.

At the core of our method for exploring the integrated schemas that result from the above choices (Steps 2 and 3 in Figure 4.1), there are three novel components that we have developed. First, we have developed an algorithm for generating one integrated schema, given a fixed choice of which matching concepts to merge. Second, the space of candidate schemas is then defined via an enumeration procedure that takes into account all possible merging choices (rather than a fixed one). This enumeration procedure provides the basis for more efficient and more directed ways of exploring the space of candidate schemas. In this chapter, we describe one such directed method (our third component) that explores only a selected set of candidate schemas, interactively, based on user-specified constraints, ultimately leading to the desired integrated schema.

Generating one Integrated Schema We describe an algorithm (ApplyAssignment) that produces a single integrated schema, given a fixed choice of which concepts to merge. This schema preserves, in a precise sense, all the attributes and relationships of the source schemas. The algorithm includes an interactive feature that allows the users to specify how to merge redundant relationships that may arise in the process. At the same time, the algorithm generates a mapping from the source schemas to the integrated schema that has precise information-preserving properties. The mapping generation component is in the spirit of the more general algorithms of mapping systems such as Clio [77], which construct mappings between independently designed. However, our mapping algorithm is more direct and has no ambiguity, by taking full advantage of how the integrated schema is generated from the source schemas.

Conceptual Enumeration of the Candidate Schemas We describe an enumeration algorithm
that can systematically generate all possible integrated schemas (and the associated mappings), by considering all possible choices of which concepts to merge. An essential feature of the enumeration algorithm is that it avoids exploring different configurations that yield the same schema. This duplication-free algorithm makes use, as a subroutine, of a polynomial-delay algorithm by Creignou and Hébrard [36] for the enumeration of all Boolean vectors satisfying a set of Horn clauses. Polynomial-delay [58] means that the delay between generating any two consecutive outputs (satisfying assignments in this case) is bounded by a polynomial in the size of the input. In a precise sense, this is the best one can hope for when the number of outputs is exponential in the size of the input.

**Interactive Exploration of the Space of Schemas** The users of our system do not need to see all the candidate schemas. Moreover, the system does not need to generate (a priori) all the candidate schemas. The full enumeration of all such schemas is not viable, even if duplicates are avoided. In this chapter, we describe an adaptation of the enumeration algorithm described above so that schemas are generated on demand and in combination with user interaction. Users can browse through the schemas that were generated so far and can also request the generation of the next integrated schema. More interestingly, users can specify constraints on the merging process itself, based on the schemas they have already seen. User constraints can be given through a visual interface, directly in terms of the concepts and of the matchings between them. For example, the user can enforce a pair (or a group) of matching concepts to be always merged or never merged. Additionally, the user can give constraints that enforce the preservation of certain structural patterns in the input schemas. Although constraints are added primarily for semantic reasons (to incorporate the domain knowledge of the expert), they also have the benefit
of greatly reducing the space of candidate schemas. Each constraint can cut the space by as much as half. The result is an adaptive and interactive enumeration procedure that significantly reduces the space of alternative schemas explored by the user, and facilitates the selection of the final integrated schema.

We have implemented our schema integration methodology in the Clio system [56]. This chapter describes the results of an empirical evaluation of our system with both synthetic and real datasets. The experimental results demonstrate the performance of the duplicate-free enumeration algorithm, in the sense that a user of the system experiences only a small delay (less than one second) before the system generates the next integrated schema. Furthermore, the experimental results demonstrate the feasibility of the adaptive enumeration algorithm based on user constraints. Although the space of all integrated schemas can be large, the number of constraints a user specified before the system generated the desired integrated schema was small, in all scenarios used in the experiments.

4.1 Integration through Concepts

This section introduces the basic ingredients in our schema integration framework, namely the graphs of concepts. It also explains how the schema integration problem is recast into a problem of merging graphs of concepts.

We first describe the schemas and correspondences that can be input to our method by means of an example. Consider the source schemas $S_1$ and $S_2$ depicted in Figure 4.2. The schemas are represented using the nested relational (NR) model which is used as a common ab-
straction for both relational and XML schemas in our system. The NR model has been described in Section 3.3.1 and a short overview is given below, for convenience.

A nested relational schema [77] consists of a set of labels (or roots), each with an associated set type $\text{Set } \tau$, where $\tau$ is defined by: $\tau := b | \text{Rcd}[l_1 : \tau_1, \ldots, l_n : \tau_n] | \text{SetOf } \tau$. Here, $b$ represents an atomic type such as string or integer, $\text{SetOf } \tau$ is a set type, while $\text{Rcd}[l_1 : \tau_1, \ldots, l_n : \tau_n]$ is a record type where $l_1, \ldots, l_n$ are labels (or elements). Note that this is only a simplified abstraction of the NR model described in Section 3.3.1. However, our system implementation handles the NR model in its full generality (including choice types, optional and nullable elements). Figure 4.2 does not show the atomic types (i.e., the types of the leaf-level elements), for simplicity. We shall often use the term attributes for the atomic type elements. In general, attributes in a source schema are all assumed to have a unique identity (for example, the full path in the schema can be used for the purpose of identification). However, in examples, we shall often just use the name of the leaf-level element to refer to an attribute, whenever there is no confusion.

The first schema shown in Figure 4.2 represents departments with their employees and grants, as well as the projects for which grants are awarded. The depts root is a set of department records, each with three atomic components and a set-valued component, managers, which represents a (nested) set of manager records. The arrows in schema $S_1$ represent foreign key constraints: a grant has references to both a department and a project, while an employee has a reference to a department. The second schema includes a set of organization records with nested sets of locations, employees and funds. The information about managers has been condensed in this schema to one field (mgr_eid in a fund record). Moreover, in this schema,
employees have phones and each fund has a sponsoring organization (represented by a foreign key constraint) which may be different from the parent organization.

Figure 4.2 also shows correspondences between atomic type elements (attributes) of the schemas $S_1$ and $S_2$. These correspondences are bi-directional and signify “equivalent” attributes (i.e., that can carry the same data) in the two schemas. They can be specified by the user or discovered through schema matching techniques. We only consider correspondences between attributes, since these are the elements that carry actual data. Note that there can be attributes with no correspondences and also attributes with multiple correspondences. For example, $\text{mgr\_eid}$ in $S_1$ matches both $\text{mgr\_eid}$ and $\text{eid}$ in $S_2$ (possibly with less confidence for the second one; in this paper, we ignore weights on correspondences and, instead, treat all correspondences the same).

**Schema Integration Desiderata** Assume that we are given $n$ source schemas $S_1, \ldots, S_n$ and
a set of correspondences that relate pairs of attributes in these schemas. We would like to compute an integrated target schema $T$ and a set $M$ of mappings from the source schemas to the integrated schema, such that $T$ and $M$ satisfy the following informal requirements:

1. The integrated schema $T$ is capable of representing all the atomic-type information in the source schemas, in the sense that every attribute occurring in $S_1, \ldots, S_n$ must be represented in $T$. However, and this will often be the case, it is possible that one target attribute may represent multiple source attributes that are “equivalent” according to the correspondences.

2. The integrated schema $T$ does not represent any extra atomic-type information not present in the sources, in the sense that every attribute of $T$ must represent some attribute of $S_1, \ldots, S_n$.

3. Every tuple and every join\(^1\) of tuples that can be present in a source database conforming to a source schema is “transferred” via $M$ into a similar tuple or join of tuples under the integrated schema. In a sense, we require the preservation of all the basic relationships that can exist in each of the sources. All the necessary notions shall be formally defined in Section 4.2.2 which describes the mapping generation component of the system.

We now embark on the exposition of the main steps towards achieving such integrated schema $T$ and set $M$ of mappings.

### 4.1.1 Concepts

The first key idea is to abstract the concrete physical layout of schemas into a more logical view that is based on concepts. Each schema (with constraints and with nesting) can be

\(^1\)Either parent-child type of join or foreign key / key type of join.
replaced by a graph of flat concepts where the edges represent HasA relationships. Formally, a concept graph can be defined (independently of a schema) as follows. We fix \( \mathcal{U} \) to be a universe of attributes. We also fix \( \mathcal{L} \) to be a universe of labels (different from attributes).

**Definition 4.1.1 (Concept graph)** A concept is a relation name \( C \) associated with a subset \( \text{att}(C) \) of \( \mathcal{U} \) (these are the attributes of \( C \)). A concept graph is a pair \( (V, \text{HasA}) \) where \( V \) is a set of concepts and \( \text{HasA} \) is a set of directed edges between concepts, such that each edge has a label from \( \mathcal{L} \). We write \( A \text{HasA} B \ [L] \) whenever there is a \( \text{HasA} \) edge with label \( L \) from concept \( A \) to concept \( B \).

Intuitively, the meaning behind \( A \text{HasA} B \ [L] \) is that every instance of concept \( A \) has a reference (of type \( L \)) to exactly one instance of concept \( B \). The role of the \( \text{HasA} \) edges is to express that certain concepts cannot exist without other concepts (they extend or depend on those concepts). Also, another way to view an edge \( A \text{HasA} B \ [L] \) is that it represents a many-to-one relationship from \( A \) to \( B \): there can be zero or more \( A \) instances with references to the same \( B \) instance. Note that, in general, there could be more than one \( \text{HasA} \) edge between two concepts and, moreover, the graph can have cycles.
Figure 4.3 shows the two concept graphs that “correspond” to the schemas $S_1$ and $S_2$. Although each edge has a label (which we assume is either system generated or given by the user), we may sometimes drop the label whenever it is not important. Note that in the case of parallel $\text{HasA}$ edges (e.g., the two $\text{HasA}$ edges labeled $\text{sponsor}$ and $\text{owner}$ in the second concept graph), the labels are needed in order to be able to distinguish between these edges.

To give the intuition of how these concept graphs relate to the schemas, consider the concept graph for $S_1$. There, dept and project are top-level concepts (i.e., have no outgoing $\text{HasA}$ edges), corresponding to the top-level sets $\text{depts}$ and $\text{projects}$. These are standalone concepts that do not depend on anything else. In contrast, there is a $\text{HasA}$ edge from manager to dept, since a manager element cannot exist independently of a department (according to the nesting in $S_1$).

Similarly, there is a $\text{HasA}$ edge from emp to dept, reflecting the fact that, in $S_1$, an employee has a foreign key to a department. Also, grant has edges to both dept and project, since a grant has foreign keys into both department and project. More interestingly, the concept graph for $S_2$ includes two parallel $\text{HasA}$ edges from fund to org, reflecting the fact that a fund is nested under a parent organization (the “owner”) and also a fund has a reference to a sponsoring organization (the “sponsor”). Thus, org plays a dual role with respect to fund.

Note that concepts in one concept graph are not intended to represent absolute concepts. For example, location in the second concept graph in Figure 4.3 represents the “location of an organization” and not a general notion of location. Similarly, emp represents the notion of an “employee within an organization” and not a general employee. Thus, concepts within one concept graph reflect a particular way of modeling the data. However, as schemas are in-
tegrated, concepts from different schemas can be merged and, thus, accumulate features (e.g.,
attributes and relationships) from the different schemas.

### 4.1.2 Extracting Concepts from Schemas

In order to extract the concepts and the relationships that are implicit in a schema, we
use a simple algorithm that creates one concept for each set-type element in the schema and
then uses the structure and the constraints in the schema to establish the relationships (HasA
edges) between concepts.

More concretely, for each set-type (collection) element $S$ in the schema, a concept $C_S$
is computed such that: (1) $C_S$ includes all the attributes under $S$ (without attributes from any
other set type elements that may be nested under $S$), (2) $C_S$ has a HasA edge to the concept $C_{S_1}$
encoding the parent collection $S_1$ (if such parent exists), and (3) $C_S$ has a HasA edge to $C_{S_2}$
whenever $C_S$ has an attribute that is a foreign key referring to one of the attributes (the key) of
$C_{S_2}$. Furthermore, whenever case (3) applies, the foreign key attribute from the list of attributes
of $C_S$ is dropped (since it is represented at $C_{S_2}$). In both cases (2) and (3) a fresh label is
computed for the HasA edge. Case (3) also applies, with slight changes, when the foreign keys
(and keys) are composite.

The concept graphs resulting from the two source schemas $S_1$ and $S_2$ are shown in
Figure 4.3. Note that the name given to a concept or to a HasA edge is not essential. A possible
choice for concepts is to generate a name based on the set-type element from which the concept
is constructed. For HasA edges that are derived from foreign keys, the name of the foreign key
attribute could be used.
We note at this point that it is necessary to keep track of the implicit mapping from a schema to its concept graph. In particular, for each concept, we remember which set type element it corresponds to, and for each HasA edge, we remember either the parent-child relationship or the foreign key constraint it was generated from. This mapping is necessary to be able to translate any subsequent mappings that are expressed in terms of the concepts back in terms of the input schemas.

Finally, the algorithm can deal with cyclic integrity constraints by simply transferring them into cycles over the HasA edges. (For example, the resulting concept graph may include Dept HasA Emp [manager] and Emp HasA Dept [works_for]).

Once the extraction of concepts is achieved, most of the subsequent processing (including user interaction) is performed at the level of concepts and not schemas. This is beneficial since concepts are simpler and also reflect better the logical meaning behind the schemas. Nonetheless, once the concepts are integrated, we can go back to create an integrated schema, as described in Section 4.2.3. We note that our subsequent integration method can take as input arbitrary concept graphs that are not necessarily the result of extraction from schemas. Thus, it can be applied directly to any logical models as long as they can be expressed as concept graphs with HasA relationships.

### 4.1.3 Matching the Concepts

This section describes how the input correspondences that are given in terms of schemas are translated into “matching” edges between the concepts that correspond to the schemas. The result of this translation is a matching graph that will be the main object of
the subsequent processing (i.e., the actual merging and enumeration).

**Definition 4.1.2** Let \( S_1 \) and \( S_2 \) be two source schemas and let \( C \) be a set of correspondences between attributes of \( S_1 \) and \( S_2 \). Let \( A \) be a concept of \( S_1 \) and \( B \) be a concept of \( S_2 \). We say that \( A \) and \( B \) match if there is at least one attribute \( a \) in \( A \) and one attribute \( b \) in \( B \) such that there is a correspondence at the schema level between attribute \( a \) and attribute \( b \).

Matching concepts will be the candidates for merging. The notion of a matching graph is described next. In the matching graph, the nodes are the concepts (in all the schemas), while the edges indicate matching concepts. Additionally, the matching graph also records the HasA edges, which will be distinguished from the matching edges.

**Definition 4.1.3 (Matching graph)** Let \( S_1, \ldots, S_n \) be schemas and let \( C \) be a set of correspondences between attributes of these schemas. The matching graph associated with \( S_1, \ldots, S_n \) and \( C \) is an undirected graph \( G = (V, \text{HasA}, E) \) where:

- The set \( V \) of nodes is the set of concepts of \( S_1, \ldots, S_n \);
- The set \( \text{HasA} \) is the union of the sets of HasA edges obtained from the individual schemas;
- The set \( E \) of edges contains exactly one edge for each pair of matching concepts in \( V \).

Figure 4.4 shows the matching graph \( G \) for our example. (Note that for two schemas, the matching graph is a bipartite graph.) For simplicity, the HasA relationships are not shown as edges, to avoid cluttering. Instead, they are written as part of the concepts themselves. For example, a statement HasA dept is added to the manager concept definition to denote that there is a HasA edge from manager to dept. (Notice that, at this point, a HasA edge never relates concepts that come from different schemas.) In the figure, the concepts in the two schemas are
aligned so that matching concepts are close to each other. The layout used in the figure is very similar to the way matching concepts are illustrated in our system (Section 4.5), which uses specialized graph-displaying packages to help in visualizing the matching.

The matching edges in Figure 4.4 are denoted as $x_0$ to $x_7$. In general, an edge $x$ between concepts $A$ and $B$ may exist because of multiple pairs of attributes $(a, b)$ that satisfy the condition in Definition 4.1.2. For example, $x_0$ exists due to the pairs (dno, oid) and (dname, oname). Nevertheless, only one edge is added between such concepts.

In general, one concept may match with multiple concepts in other schemas. For example, dept matches with org but also with location, since the attribute country in dept has
a correspondence to attribute country in location. A priori, we should not assume that dept may match “better” with org than with location. In fact, it may happen that location is meant to represent a branch of an organization, and dept in the first schema also has the meaning of a branch, in which case dept matches better with location. As another example of multiple matchings, the concepts of manager and emp in the first schema match with both emp and fund in the second schema.

The enumeration algorithm (Section 4.4) will take into account all choices of merging and will explore all possible ways of integration. Note that there may also be concepts that do not match with any concept. For example, phone is specific to the second schema. Even though they have no matches, such concepts are still involved in the merging process, since they may have HasA edges to other concepts.

### 4.2 Merging the Concepts

**Assignments** Different ways of merging the concepts can be obtained by considering different subsets of edges in the matching graph $G$. Let $A$ and $B$ be two matching concepts and let $x$ denote the edge between these concepts in $G$. We can think of the edge $x$ as having a value of 0 or 1: $x = 0$ means that $A$ and $B$ need not be merged (i.e., the edge can be ignored), while $x = 1$ means that $A$ and $B$ must be merged (i.e., the edge must be applied). Every Boolean assignment $X$ to the set $E$ of edges in $G$ will yield an integrated concept graph which in turn will result in an integrated schema. The following are two assignments for the edges in our example:

$$X_1: \{ x_1 = x_2 = x_3 = x_5 = 0, x_0 = x_4 = x_6 = x_7 = 1 \}$$
$$X_2: \{ x_0 = x_3 = x_5 = 0, x_1 = x_2 = x_4 = x_6 = x_7 = 1 \}$$
The first assignment requires dept in the first schema to be merged with org in the second schema, emp in the first schema to be merged with emp in the second schema, and grant and project in the first schema to be merged with fund in the second schema. Under the second assignment, we must merge dept with location, manager with the two emp concepts, and grant and project with fund.

An assignment $X$ shall sometimes be identified with the subset $E_X$ of edges that have the value 1 under the assignment $X$.

This section describes the ApplyAssignment algorithm which takes a matching graph and one assignment for the edges, and produces one integrated concept graph. At the same time, the algorithm also generates the mapping from the source concepts to the integrated concepts that specifies how source data has to be transformed into the integrated data. Section 4.4 describes in detail the procedure for enumerating multiple, distinct, integration results, by repeatedly invoking ApplyAssignment on different assignments.

### 4.2.1 The ApplyAssignment Algorithm

ApplyAssignment($G, X$)

**Input:** Matching graph $G = (V, HasA, E)$, Boolean assignment $X$ for $E$.

**Output:** Integrated concept graph $G' = (V', HasA')$, mapping $M$ between source and integrated concept graphs.

Let $E_X$ be the subset of edges that have been assigned the value 1.

1. Create the integrated concepts.
(a) Compute the connected components in the graph $G_X = (V, E_X)$.

(b) For every connected component $[A_1, \ldots, A_k]$ of $G_X$, where $A_1, \ldots, A_k$ are concepts in $V$, create an integrated concept $C$ (i.e., a node in $V'$). Let $\text{att}(C)$ be the union of the attributes in $A_1, \ldots, A_k$, where corresponding attributes are considered duplicates and are represented only once.

For every $A_i$ among $A_1, \ldots, A_k$, define $f_X(A_i) = C$. Furthermore, for every attribute $a$ of $A_i$, define $f_X(A_i.a) = C.a^*$, where $a^*$ is the representative for the group of duplicates of $a$.

2. Construct HasA edges between integrated concepts. For every $A$ HasA $B [L]$ in $G$, create $f_X(A)$ HasA $f_X(B) [L]$ in $G'$ (i.e., an edge in HasA'); define $f_X(A \text{ HasA } B [L]) = f_X(A) \text{ HasA } f_X(B) [L]$.


4. Create mapping $\mathcal{M} = \text{MapGen}(G, G', f_X)$.

At the high-level, the algorithm ApplyAssignment described above first applies all the mergings between concepts that are required by the input assignment. This is Step 1 of the algorithm. Step 2 creates HasA relationships among the integrated concepts, based on the source HasA relationships. The user is then allowed to refine the resulting integrated concept graph in Step 3. In the process, ApplyAssignment maintains an integration function $f_X$ (for the given assignment $X$) that specifies how each individual attribute, concept or HasA edge in a source concept graph relates to a corresponding attribute, concept, and, respectively, path of HasA edges, in the integrated concept graph. We shall elaborate on the use of the integration function in Section 4.2.2, which explains the construction of the mapping between the source
and integrated concept graphs in Step 4 of *ApplyAssignment*.

### 4.2.1.1 Step 1: Connected Components

First, we compute the connected components in the graph $G_X = (V, E_X)$ that is induced from the matching graph $G$ by considering only the edges in $X$ with value 1. For each connected component, an integrated concept is obtained by taking the union of the attributes of the source concepts in that connected component, while at the same time collapsing duplicate
attributes. Two attributes $a$ and $b$ of source concepts in a connected component are considered *duplicates* if: (1) there exists a correspondence between $a$ and $b$ at the schema level, or (2) there exists another attribute $c$ of a source concept in the same connected component such that $c$ is a duplicate of both $a$ and $b$. For every group of duplicates we then pick, arbitrarily from the group, a representative that will be subsequently used, in the integrated concept, in place of the individual attributes in the group. As a convention (to signify that a choice was made), the name of the representative is suffixed with the symbol “*”, whenever there are at least two duplicates in the group.

Figure 4.5(a) shows the integrated concepts that result after Step 1 of ApplyAssignment, when given the assignment $X_1$ shown earlier. (Ignore the HasA edges between these concepts for now.) There are six integrated concepts, corresponding to the six connected components in the graph $G_{X_1} = (V, E_{X_1})$, where $E_{X_1} = \{x_0, x_4, x_6, x_7\}$. For example, the integrated concept denoted as $[\text{dept, org}]$ corresponds to the connected component consisting of the source concepts dept and org. The union of the attributes in dept and org is $\{\text{dno, dname, country, oid, oname}\}$. However, dno and oid are duplicates and are replaced by a unique occurrence of their representative (chosen as dno*). Similarly, dname and oname are replaced by the representative dname*.

The relationship between the source concepts and the integrated concepts is recorded via the *integration function* $f_X$. For simplicity, we may write $f$ instead of $f_X$, if the assignment $X$ is understood from the context. Concretely, each source concept $C$ is mapped into the integrated concept that $C$ is merged into. For example, we have:
Moreover, each attribute of a source concept $C$ is mapped into the representative attribute in the integrated concept $f(C)$. For example, we have:

\[
\begin{align*}
  f(\text{dept}. \text{dno}) &= f(\text{org}. \text{oid}) = [\text{dept}, \text{org}]. \text{dno}^*, \\
  f(\text{dept}. \text{dname}) &= f(\text{org}. \text{name}) = [\text{dept}, \text{org}]. \text{name}^*, \\
  f(\text{location}. \text{country}) &= [\text{location}]. \text{country}, \ldots
\end{align*}
\]

4.2.1.2 Step 2: Copying the Relationships

In this step, ApplyAssignment “copies” all source \texttt{HasA} edges into \texttt{HasA} edges on the integrated concepts. Specifically, for each source relationship $A \texttt{HasA} B [L]$, an edge $f(A) \texttt{HasA} f(B) [L]$ is created between the integrated concepts $f(A)$ and $f(B)$. At the same time, the correspondence between the source \texttt{HasA} edge and the integrated one is recorded as $f(A \texttt{HasA} B [L]) = f(A) \texttt{HasA} f(B) [L]$.

Figure 4.5(a) shows the integrated concept graph resulting after Step 2 of the algorithm with assignment $X_1$. It is easy to see that every source \texttt{HasA} edge has a distinct corresponding \texttt{HasA} edge in the integrated graph. The integration function is now enriched with entries such as:

\[
\begin{align*}
  f(\text{location} \texttt{HasA} \text{dept}) &= [\text{location}] \texttt{HasA} [\text{dept}, \text{org}] \\
  f(\text{emp}_1 \texttt{HasA} \text{dept} [\text{default}_2]) &= [\text{emp}_1, \text{emp}_2] \texttt{HasA} [\text{dept}, \text{org}] [\text{default}_2] \\
  f(\text{emp}_2 \texttt{HasA} \text{org} [\text{default}_5]) &= [\text{emp}_1, \text{emp}_2] \texttt{HasA} [\text{dept}, \text{org}] [\text{default}_5]
\end{align*}
\]

where we write $\text{emp}_1$ for the $\text{emp}$ concept in the first schema and $\text{emp}_2$ for the $\text{emp}$ concept in the second schema.

As a result of Step 2, the integrated graph may contain parallel \texttt{HasA} edges (see the edges between $[\text{emp}_1, \text{emp}_2]$ and $[\text{dept}, \text{org}]$) as well as \texttt{HasA} loops (see the self-loop on
In general, parallel edges must be considered different, since we cannot assume, without additional knowledge, that the relationships encoded by them are the same. For example, we cannot automatically assume that the relationship between an employee and a department that is coming from the first schema is the same as the relationship between an employee and an organization that is coming from the second schema. (See also owner and sponsor as an example of two parallel edges that represent different relationships.)

A similar argument prevents us from automatically removing loops: the relationship between the grant part of [grant, fund, project] and the project part of [grant, fund, project] that is now implicit in the fact that the two concepts have been merged may not be the same as the original source relationship between grant and project. We may need both, in which case we have to keep the original edge as a loop.

Hence, Step 2 of the algorithm will include by default all the HasA relationships between integrated concepts that can be derived from the source HasA relationships.

### 4.2.1.3 Step 3: Removal of Redundant Relationships

This step of ApplyAssignment allows the user to interactively merge parallel edges and remove loops in the integrated graph, whenever the user deems them as redundant. To represent the user feedback, we use a special form of constraints that allows us to remember the information and re-apply it in subsequent invocations of ApplyAssignment (for different assignments). Since these constraints are used to specify redundant information, we call them redundancy constraints. These are constraints on the design of the integrated schema (and not constraints on the data).
Parallel HasA edges can be merged by using redundancy constraints of the form:

\[
\begin{align*}
\text{if} & \quad f(A) = f(A'), f(B) = f(B') \\
\text{then} & \quad f(A \text{ HasA } B [L]) = f(A' \text{ HasA } B' [L'])
\end{align*}
\]

where \( A \text{ HasA } B [L] \) and \( A' \text{ HasA } B' [L'] \) are two source HasA edges. The meaning of such constraint is that, in any integrated graph where we merge \( A \) and \( A' \) (i.e., \( f(A) = f(A') \)) and we also merge \( B \) and \( B' \) (i.e., \( f(B) = f(B') \)), the two parallel HasA edges with labels \( L \) and \( L' \) that result in the integrated graph must be considered equal. In a sense, the constraint says that the two source relationships \( A \text{ HasA } B [L] \) and \( A' \text{ HasA } B' [L'] \) are equivalent, whenever the concepts involved in the relationships are merged, pairwise.

To enforce such constraint, one of the two parallel edges must be removed. We take the convention that the first edge will always be removed in favor of the second. As a result of applying the constraint, the integration function \( f \) is also updated so that the two source edges map both into the surviving edge in the integrated graph.

For the integrated concept graph in Figure 4.5(a), a user may state the following constraint:

\[
(C_1) \quad \begin{align*}
\text{if} & \quad f(\text{emp}_2) = f(\text{emp}_1), f(\text{org}) = f(\text{dept}) \\
\text{then} & \quad f(\text{emp}_2 \text{ HasA } \text{org} [\text{default}_5]) = f(\text{emp}_1 \text{ HasA } \text{dept} [\text{default}_2])
\end{align*}
\]

to remove the edge from \([\text{emp}_1,\text{emp}_2]\) to \([\text{dept,org}]\) that is labeled \( \text{default}_5 \) in favor of the edge labeled \( \text{default}_2 \) (see Figure 4.5(b)). A similar constraint can express the merging of the two parallel edges between \([\text{grant,fund,project}]\) and \([\text{dept,org}]\) that are labeled \( \text{default}_3 \) and \( \text{owner} \) into one edge labeled \( \text{owner} \).

Note that such constraints will also apply in other integrated concept graphs, as long as the premises of the constraints are satisfied. For example, assume that later on, the system
generates, based on some other assignment, a different integrated graph where \( \text{emp}_1 \) and \( \text{emp}_2 \) are still merged (possibly with some other concepts, like manager), and where \( \text{dept} \) and \( \text{org} \) are still merged (possibly with some other concepts, like location). Then the user does not have to restate that \( \text{emp}_1 \; \text{HasA} \; \text{dept} \) and \( \text{emp}_2 \; \text{HasA} \; \text{org} \) are the same relationship.

Loops in the integrated graph can be eliminated by means of redundancy constraints of a slightly simpler form:

\[
\text{if } f(A) = f(B) \quad \text{then } f(A \; \text{HasA} \; B \; [L]) = f(A)
\]

In the above, \( A \; \text{HasA} \; B \; [L] \) is a source \( \text{HasA} \) edge. The meaning of the constraint is that, in any integrated graph where we merge \( A \) and \( B \) (i.e., \( f(A) = f(B) \)), the resulting loop \( f(A) \; \text{HasA} \; f(A) \; [L] \) must be removed from the integrated graph, and the source \( \text{HasA} \) edge must be represented by \( f(A) \).

Based on the integrated concept graph in Figure 4.5(a), a user may state the following constraint:

\[
(C_2) \quad \text{if } f(\text{grant}) = f(\text{project}) \quad \text{then } f(\text{grant} \; \text{HasA} \; \text{project} \; [\text{default}_4]) = f(\text{grant})
\]

to remove the loop labeled \( \text{default}_4 \) (as in Figure 4.5(b)). As a result, the relationship between grant data and project data is encoded directly within the integrated concept [grant,fund,project] (within one tuple). The integration function is changed accordingly so that the source \( \text{HasA} \) edge from grant to project is mapped into the single concept [grant,fund,project].

From a user interaction point of view, we note that the constraints can be discovered by visualizing the parallel edges or loops that arise in an integrated concept graph. In the
visual interface of the system, the cause for such parallel edges or loops is traced back to the
sources (via the integration function \( f \)). The user can then immediately state the “desired”
constraints in terms of the source edges (e.g., state that two source edges are equivalent, or that
one source edge should be collapsed whenever it becomes a loop). We view the mechanism of
user constraints (redundancy constraints here, and enumeration constraints later in Section 4.4)
as a form of learning domain knowledge from a user. Once such knowledge is learned, it is
automatically reapplied in other configurations, during the same run of the system.

4.2.1.4 The General Form of Redundancy Constraints

More generally, we allow users to specify constraints that map a source \texttt{HasA} edge
into a path of zero, one or more \texttt{HasA} edges in the integrated concept graph. We have seen
eamples where a source \texttt{HasA} edge is mapped into a path of length zero (in the case of \( C_2 \))
or of length one (in the case of \( C_1 \)). The following example illustrates a case where a source
\texttt{HasA} edge needs to be mapped into a path of two edges in the integrated graph. For brevity,
we shall use the notation \( A \xrightarrow{L_1} B \xrightarrow{L_2} C \) to represent a path of edges \( A \texttt{HasA} B [L_1] \) and
\( B \texttt{HasA} C [L_2] \).

Consider the integrated graph in Figure 4.6, which is obtained by \texttt{ApplyAssignment}
when given the earlier assignment \( X_2 \) (instead of \( X_1 \)) and the constraints in Section 4.2.1.3.
The following constraint:

\[
\text{if } f(\text{emp}_2) = f(\text{emp}_1), f(\text{dept}) = f(\text{location}) \quad \text{then } f(\text{emp}_2 \texttt{HasA} \text{org } [\text{default}_5]) = f(\text{emp}_1) \xrightarrow{\text{default}_2} f(\text{dept}) \rightarrow f(\text{org})
\]

implies that the edge from \( \text{[manager,emp}_1,\text{emp}_2] \) to \( \text{[org]} \) that is labeled \texttt{default}_5 is made

193
Figure 4.6: Another integrated concept graph.

Redundant by the path \([\text{manager, emp}_1, \text{emp}_2] \xrightarrow{\text{default}_2} [\text{dept, location}] \rightarrow [\text{org}]\) and can be removed. The constraint asserts that, in general, whenever we merge \(\text{emp}_2\) with \(\text{emp}_1\), and \(\text{dept}\) with \(\text{location}\), the source relationship \(\text{emp}_2 \text{HasA} \text{org} [\text{default}_3]\) is equivalent to the relationship implied by the two source edges \(\text{emp}_1 \text{HasA} \text{dept} [\text{default}_2]\) and \(\text{location} \text{HasA} \text{org}\). Note that the latter source edges are not in the same source schema and do not form a path. However, the condition \(f(\text{dept}) = f(\text{location})\) in the premise of the constraint implies that their images (under \(f\)) form a path in the integrated graph.

Redundancy constraints have the following general form:

\[
\text{if } X \text{HasA} Y [L], \\
A_0 \text{HasA} A_1 [L_1], A_1' \text{HasA} A_2 [L_2], \ldots, A_{n-1}' \text{HasA} A_n [L_n], \\
f(X) = f(A_0), f(Y) = f(A_n), f(A_k) = f(A_k'), 1 \leq k \leq n \\
\text{then } f(X \text{HasA} Y [L]) = f(A_0) \xrightarrow{L_1} f(A_1) \xrightarrow{L_2} \ldots \xrightarrow{L_n} f(A_n)
\]

where \(X, Y, A_0\) to \(A_n\) and \(A'_1\) to \(A'_{n-1}\) are source concepts and \(f(A_{i-1}) \xrightarrow{L_i} f(A_i)\) denotes \(f(A_{i-1}) \text{HasA} f(A_i) [L_i]\). Note that the edges involving the “A” concepts need not be in the
same source schema. These edges may not form a path, since they can be in different graphs; however, the premise of the constraint implies that their images (under \( f \)) form a path in the integrated graph and, moreover, that the end-points of the path are merged with \( X \) and \( Y \), respectively. The meaning of the constraint is that the edge from \( X \) to \( Y \) is redundant in the integrated graph (its semantics is implied by the relationship on the path) and it will be removed.

From a user’s perspective, it is clearly difficult to discover redundancy constraints that map a source HasA into a path of length more than one in the integrated graph, without additional help. In the visual interface of our system, we provide a simple mechanism for suggesting possible redundancy constraints that can be specified for a given edge \( X \text{ HasA } Y \) in the source graph. This mechanism is based on computing all paths from \( f(X) \) to \( f(Y) \) in the integrated graph. For each path, the system suggests the corresponding constraint to the user by highlighting the edges (in the source and integrated graphs) involved in the constraint.

### 4.2.2 Mapping Generation

This section describes the mapping generation algorithm \( \text{MapGen} \), which takes as input the matching graph \( G \), the integrated concept graph \( G' \) and the integration function \( f \) resulting after Step 3 of \( \text{ApplyAssignment} \), and outputs a mapping \( M \) between the source and integrated concepts. Specifically, for every source concept \( C \), a mapping \( M_C \) is added in \( M \). The mapping \( M_C \) specifies how an instance of \( C \), together with all its relationships, is to be transformed into an instance (possibly associated with other instances) of an integrated concept \( C' \). The mappings generated by \( \text{MapGen} \) are in the spirit of the nested schema mappings formalism described in Section 3.3.1. Our system translates the generated mappings from the level
of concepts back to the level of schemas, as described in Section 4.2.3. The resulting mappings between the source schemas and the integrated schema are expressed using the nested schema mappings formalism. MapGen is described below and it is illustrated with an example next.

\textbf{MapGen}(G, G', f)

\textbf{Input:} Matching graph \( G = (V, \text{HasA}, E) \), integrated concept graph \( G' = (V', \text{HasA}') \), and integration function \( f \).

\textbf{Output:} Mapping \( \mathcal{M} \) from \( G \) to \( G' \).

For every source concept \( C \) of \( G \) do:

1. Initialize \( \mathcal{M}_C \) to be

   \[
   \text{for } c \in C \text{ exists } c' \in C' \text{ where } \bigwedge_{a \in \text{att}(C)} (c.a = c'.a^s),
   \]

   provided that \( f(C) = C' \) and \( f(C.a) = C'.a^s \), for each \( a \in \text{att}(C) \).

   Initialize queue \( Q = \{(c \in C, c' \in C')\} \).

2. Extend \( \mathcal{M}_C \) by doing a “parallel chase” with \( \text{HasA} \) edges starting from \( C \). Repeat steps (a)–(b) until \( Q \) is empty.

(a) Take out \((u \in C_i, u' \in C'_j)\) from the first position in \( Q \).

(b) For every edge \( C_i \text{ HasA } C_j \text{ [L]} \) do:

   \begin{itemize}
   \item Let \( f(C_i \text{ HasA } C_j \text{ [L]}) = C'_i \xleftarrow{L_1} D_1 \xleftarrow{L_2} \ldots \xleftarrow{L_n} C'_j \).
   \item Chase \( u \in C_i \) of \( \mathcal{M}_C \) with the edge \( C_i \text{ HasA } C_j \text{ [L]} \) by adding, in the for clause, a new variable \( w \in C_j \) and join condition \( u \text{ HasA } w \text{ [L]} \).
   \item If \( n > 0 \), chase \( w' \in C'_j \) of \( \mathcal{M}_C \) with the path \( C'_i \xleftarrow{L_1} D_1 \xleftarrow{L_2} \ldots \xleftarrow{L_n} C'_j \), by adding, in the exists clause, a sequence of variables \( w_1 \in D_1, \ldots, w_n \in C'_j \), and join conditions \( w' \text{ HasA } w_1 \text{ [L]}, \ldots, w_{n-1} \text{ HasA } w_n \text{ [L]} \).
   \end{itemize}
- If \( n > 0 \), add \( \bigwedge_{b \in \text{att}(C_j)} (w.b = w_n.b^*) \) to the \textbf{where} clause of \( \mathcal{M}_C \), provided that \( f(C_j.b) = C'_j.b^* \), for each \( b \in \text{att}(C_j) \).

  Otherwise (\( n = 0 \) and \( C'_i = C'_j \)), add \( \bigwedge_{b \in \text{att}(C_j)} (w.b = u'.b^*) \) to the \textbf{where} clause of \( \mathcal{M}_C \), provided that \( f(C_j.b) = C'_i.b^* \), for each \( b \in \text{att}(C_j) \).

- Insert \((w \text{ in } C_j, w_n \text{ in } C'_j)\) (if \( n > 0 \)) or \((w \text{ in } C_j, u \text{ in } C_i)\) (if \( n = 0 \)) in \( Q \).

Return \( \mathcal{M} = \{ \mathcal{M}_C \mid C \text{ is a source concept in } G \} \).

We next illustrate MapGen and the mappings it produces with an example. Consider the integrated graph from Figure 4.5(b). Step 1 of MapGen construct a mapping from one single source concept to one single integrated concept. The following is the mapping \( \mathcal{M}_{\text{grant}} \) constructed for the source concept \text{grant}:

\[
\text{for } \quad g \text{ in grant} \\
\text{exists } \quad g' \text{ in [grant,fund,project]}  \\
\text{where } \quad g.\text{amount} = g'.\text{amount}^*
\]

This assertion specifies that for each instance \( g \) of \text{grant}, there must exist an instance \( g' \) of [grant,fund,project] (which is \( f(\text{grant}) \)) where the value for the attribute \( \text{amount}^* \) is copied from the source attribute \( \text{amount} \) (since \( f(\text{grant}.\text{amount}) = [\text{grant,fund,project}].\text{amount}^* \)).

Step 2 is the main component of MapGen and its role is to enrich the concept-to-concept mapping established in Step 1 so that it maps \textit{groups of related instances} rather than isolated instances. To illustrate, according to the first source schema, a grant instance is associated with department and project information through the two \texttt{HasA} relationships labeled \texttt{default3} and \texttt{default4}. In general, it is desirable to carry over such data associations from the source and preserve them in the integrated data. Consequently, \( \mathcal{M}_{\text{grant}} \) is extended in Step
2 of MapGen in order to transfer, in the integrated data, all the instances that are directly or transitively associated with a grant instance.

This extension is done by a parallel chase in both the for and exists clauses of the mapping, by recursively joining in all the concepts that are related via HasA relationships. At the same time the mapping is extended on the joined concepts by using the function $f$. For our example, the following updated mapping is obtained in the iteration of Step 2(b) with the edge grant HasA dept [default3]:

```sql
for g in grant, d in dept; g HasA d [default3], exists g' in [grant,fund,project], d' in [dept,org]; g' HasA d' [owner],
where g.amount = g'.amount* and d.dno = d'.dno* and
d.dname = d'.dname* and d.country = d'.country
```

As it can be seen, the for clause is extended by adding a join with the dept concept. The notation $g$ HasA $d$ [default3] is used to express the join at the instance level between $g$ (an instance of grant) and $d$ (an instance of dept). At the same time, we add a corresponding join in the exists clause. We use the fact that $f(grant$ HasA $dept$ [default3]) = [grant,fund,project] HasA [dept,org] [owner]. Based on the integration function, we then add to the where clause of $M_{grant}$ all the equalities between the attributes of the instances $d$ of dept and $d'$ of [dept,org].

Next, in a second iteration of the same Step 2(b) of the algorithm, $M_{grant}$ is extended along the default4 relationship as follows:

```sql
for g in grant, d in dept, p in project; g HasA d [default3], g HasA p [default4], exists g' in [grant,fund,project], d' in [dept,org]; g' HasA d' [owner],
where g.amount = g'.amount* and d.dno = d'.dno* and
d.dname = d'.dname* and d.country = d'.country and
p.pid = g'.pid and p.pname = g'.pname* and p.year = g'.year
```

Here, we used the fact that $f(grant$ HasA $project$ [default4]) = [grant, fund, project].
While the for clause is extended with a join with project on default, there is no need for such extension in the exists clause. The same instance $g'$ of [grant,fund,project] that the grant instance $g$ maps into is also used to map the associated project instance $p$. We only need to add the equalities between the attributes of $p$ and the corresponding attributes of $g'$ (again, using $f$).

In the next two iterations of Step 2, the algorithm tries to extend $M_{grant}$ along any HasA edges that may be outgoing from the dept and project instances that were added to the for clause. However, dept and project are top-level concepts without any such outgoing HasA edges. Hence, Step 2 finishes at this point and $M_{grant}$ is completed.

We note that the parallel chase procedure described above does not terminate in the case of cyclic sets of constraints. To ensure that MapGen terminates (and outputs a finite mapping), we add a simple cycle detection condition that avoids further expansion based on HasA edges of concepts that have been expanded before.

**Further Remarks on Mapping Generation** The parallel chase we use here is a variation on the known chase technique [3]. The chase in the for clause of the mapping is essentially the same as the standard chase. The additional part is that we extend this chase, in parallel, by using the function $f$, on the exists clause. Therefore, the mappings transfer all the relationships that can exist in the source into corresponding relationships on the integrated schema.

Our mapping generation algorithm is directed by the function $f$ computed in the ApplyAssignment algorithm. This function dictates which concepts map to which concepts and also dictates what join conditions to use. This is in contrast with more general mapping generation algorithms of schema mapping based systems such as Clio [56, 77], which construct mappings between independently designed source and target schemas and, as such, have to
consider all possible candidate mappings between all pairs of concepts, and with all possible
choices of join conditions. The users of such systems would then have to specify which choices
to actually use (e.g., a join on owner or one on sponsor). In contrast, in our schema integration
context, the function $f$ has already encoded in it which concepts and which joins to pick. Hence,
mapping generation is more direct.

Although similar to the source-to-target tuple generating dependencies (tgds) used
for expressing schema-based mappings (i.e., mappings between relational, nested relational,
or XML schemas), the mapping constraints described in this section operate at the level of
concept graphs rather than schemas. However, the translation of these concept-level mappings
into schema mappings is straightforward, as we shall explain next.

4.2.3 From Concepts to Integrated Schema

From an integrated concept graph we can generate two types of integrated schemas:
relational or nested (XML). In the relational version, each concept is implemented in a standard
way, as a relation that includes all the attributes of the concept together with an additional key
attribute. The HasA edges are then encoded by adding appropriate foreign keys in the concepts
that have outgoing HasA edges.

In the nested version, each concept is implemented using a set-type of records con-
taining the attributes of the concept, plus a key attribute. The set-types are then nested, by using
the fact that a HasA edge represents a many-to-one relationship: if $A$ HasA $B [L]$, then there
are zero or more instances of $A$ that each have one reference (of type $L$) to one instance of
$B$. Hence, for each concept $A$, if $A$ has HasA edges to concepts $B_1, \ldots, B_m$, then we nest the
Figure 4.7: Nested schema constructed from the integrated concept graph of Figure 4.5(b).

set-type for $A$ inside the set-type for $B_1$, and add foreign key attributes referencing the keys of $B_2, \ldots, B_m$. At the end, all key attributes that are not referenced by foreign keys can be removed. (The same can be done for the relational schema representation.) The relational and nested schemas that we construct are equivalent, only the presentation/layout is different.

Figure 4.7 shows the nested schema constructed for the integrated concept graph of Figure 4.5(b). Note that the set-type [grant,fund,project] is nested under [dept,org], and at the same time, it has a foreign key into [dept,org], thus encoding the two distinct relationships with labels owner and sponsor between the corresponding integrated concepts.

After creating the integrated schema, we also create the final mapping from the input schemas to the integrated schema. Although the mappings that are generated by MapGen
are formulated in terms of concepts, translating such mappings back in terms of the concrete schemas (source and integrated) is straightforward. Essentially, all the HasA joins have to be reformulated in terms of either parent-child navigation or key / foreign key joins (depending on how the relationships are encoded in the concrete schemas). The resulting mappings are expressed using the nested schema mappings language described in Section 3.3.1. Mapping generation is the main step towards generating the actual data transformation script (or view) from the sources to the integrated schema. Once the mappings are generated, we can apply any of the existing techniques [21, 56] for compiling the mappings into the run-time queries (e.g., SQL, XQuery) that are needed to migrate the data. Once the data is migrated, additional processing is usually required to clean the integrated data.

4.3 Preservation Properties

The following proposition summarizes the main features of the ApplyAssignment algorithm (which includes the MapGen algorithm).

**Proposition 4.3.1** Let \( G = (V, \text{HasA}, E) \) be a matching graph and let \( G' = (V', \text{HasA}') \) be the integrated graph produced by \( \text{ApplyAssignment}(G, X) \), for some assignment \( X \) to \( E \). Let \( f \) be the integration function produced for \( X \).

1. Let \( A \) be a source concept. Then for every attribute \( a \) of \( A \) there is an attribute \( a^* \) in the integrated concept \( f(A) \) such that \( f(A.a) = f(A).a^* \). Conversely, let \( C \) be an integrated concept. Then for every attribute \( c \) of \( C \) there is some source concept \( A \) and some attribute \( a \) of \( A \) such that \( f(A.a) = C.c \).
2. Let $A$ \texttt{HasA} $B$ $[L]$ be a \texttt{HasA} edge in $G$. Then there is a path of zero or more \texttt{HasA} edges in $G'$ such that $f(A \texttt{HasA} B [L]) = f(A) \xrightarrow{L_1} \ldots \xrightarrow{L_n} f(B)$. Conversely, let $C$ \texttt{HasA} $D$ $[L]$ be a \texttt{HasA} edge in $G'$. Then there is an edge $A$ \texttt{HasA} $B$ $[L]$ in $G$ such that $f(A) = C$, $f(B) = D$ and $f(A \texttt{HasA} B [L]) = C \texttt{HasA} D [L]$.

3. Let $I$ be a data instance for the graph of source concepts, and let $J$ be a data instance for the integrated concept graph that is generated by enforcing, in a canonical way, all the mapping constraints produced by MapGen. Moreover, assume that there are no cycles of \texttt{HasA} edges among the source concepts.

For every instance $t$ in $I$ of a source concept $C$, there is a corresponding instance $t'$ of $f(C)$ that is generated in $J$, such that for each attribute $a$ of $t$, the value $t.a$ equals $t'.f(a)$.

Moreover, whenever such $t$ in $I$ generates a corresponding $t'$ in $J$, then for each instance $u$ in $I$ that $t$ refers to (via a \texttt{HasA} edge) there is a corresponding instance $u'$ that is generated in $J$. Furthermore, $t'$ and $u'$ are related in $J$ by a path of \texttt{HasA} edges that corresponds (via $f$) to the \texttt{HasA} edge from $t$ to $u$.

The proposition is an immediate consequence of the way the algorithms ApplyAssignment and MapGen work. The first part states that all the attributes in source concepts are transferred to attributes of integrated concepts. Moreover, there are no “new” attributes in the integrated schema. Thus, the first two informal requirements stated in Section 4.1 are satisfied in a precise sense. The second part of the proposition states that all \texttt{HasA} edges between the source concepts are transferred to paths of \texttt{HasA} edges in the integrated schema. Moreover, every \texttt{HasA} edge in the integrated schema comes from an edge (with the same label) in the source schema.
Finally, the third part of the proposition states a stronger preservation property that holds at the data level, and captures the third informal requirement in Section 4.1. It states that every instance $t$ of a source concept maps to a corresponding integrated instance $t'$. Moreover, whenever such mapping of $t$ into $t'$ takes place, we also map all the instances related to $t$ into instances related to $t'$.

The generation of a canonical integrated instance $J$ based on the mapping constraints produced by MapGen is always possible (and in polynomial time) [44, 77]. This is due to the fact that the mappings themselves have no cyclicity. However, in the case of cycles of HasA edges among the source concepts, not all the relationships can be preserved. In particular, the paths of HasA edges between source instances can be of unbounded length. Some form of recursive mappings will be needed to preserve such paths. In cases when there are no cycles of HasA edges among the integrated concepts, it can be shown that the canonical instance $J$ satisfies all the referential constraints in the integrated schema [77].

4.4 Enumeration of Alternatives

This section focuses on the problem of enumerating all possible assignments $X$ that will result in different integrated concept graphs. We start by discussing the full enumeration algorithm (Sections 4.4.1 and 4.4.2). We then explain how we make this algorithm interactive and adaptive so that only a partial enumeration is needed (Section 4.4.3).
4.4.1 Duplicates and Cycles

As discussed in Section 4.2, different assignments (such as $X_1$ and $X_2$) encode different ways of merging the input concept graphs and therefore, may give different results for the integration. A naive enumeration algorithm would exhaustively go through $2^n$ Boolean combinations, if $n$ is the size of $X$ (i.e., the number of matching edges), and then, for each combination, would run the ApplyAssignment algorithm. Besides the potential infeasibility of enumerating a large number of combinations, the naive enumeration algorithm also has the drawback that there may be many assignments that give the same integrated schema (i.e., duplicate assignments). A better approach, which we shall follow, is to avoid, from the beginning, the enumeration of duplicate assignments. As a result, a significant portion of the space of assignments can be pruned.

We shall describe next how duplicates can arise in a naive enumeration algorithm. Recall that given an assignment $X$, Step 1 in the ApplyAssignment algorithm merges together all the source concepts that are connected by the edges selected by $X$. Thus, there is one integrated concept for each connected component in the subgraph of $G$ induced by $X$. Suppose now that we include one extra edge $x$ in the set of edges that are selected by $X$. It is then possible that the effect of this extra edge $x$ is subsumed by edges that are already selected by $X$. In other words, $x$ specifies that two concepts $A$ and $B$ should be merged but this merge is already a consequence of other selected edges (i.e., there is already a path of edges selected by $X$ that connects $A$ and $B$). Thus, applying the assignment $X \cup \{x\}$ will result in the same configuration (same connected components) as applying $X$. 

205
We say that $X$ and $Y$ are duplicate assignments whenever the sets of connected components induced by $X$ and $Y$, respectively, coincide. Duplicate assignments always result in duplicate integrated graphs. Furthermore, duplicate assignments always arise due to cycles in the matching graph $G = (V, \text{HasA}, E)$. Here, the HasA edges are not relevant and we mean cycles in $(V, E)$. For example, in the earlier argument with the two duplicate assignments $X$ and $X \cup \{x\}$, the extra edge $x$ closes a cycle, since $A$ and $B$ are already connected by edges from $X$. If $G$ has no cycles then it can be easily shown that there are no duplicate assignments.

To illustrate, consider the matching graph $G$ shown in Figure 4.4. The graph consists of 10 concepts, with 8 matching edges, four of which form a cycle: $x_2$, $x_4$, $x_5$, and $x_3$. It can be seen that if an assignment contains any three edges along this cycle, then adding the fourth edge yields a duplicate assignment. Equivalently, if $X$ is an assignment that contains all four edges in the cycle, then the following assignments are all duplicates of $X$: $X - \{x_2\}$, $X - \{x_4\}$, $X - \{x_5\}$, $X - \{x_3\}$. Note that this duplication is independent of the assignments to the other edges (not in the cycle). Thus, for each of the $2^4 = 16$ (partial) assignments to the four edges not in the cycle, there are 5 assignments that have the same effect (thus, four of them are unnecessary). Accordingly, there are 16 (partial assignments for edges not in the cycle) $\times$ 4 (duplicate partial assignments to edges in the cycle) = 64 assignments that should not be considered (out of the $2^8=256$ total number of possible assignments). This duplication is even higher if there are more edges outside the cycle. Also, note that duplication happens in this example because of one cycle only.

In general, cycles appear more naturally and with higher frequency in $n$-way schema integration. In such scenarios, matching concepts will appear in multiple schemas. Furthermore,
we may have mappings between multiple pairs of these schemas, thus easily forming cycles in the resulting graph of concepts. This increased number of cycles will result in an even larger number of duplicate assignments. Section 4.6 gives synthetic examples of \( n \)-way schema integration and further illustrates the impact of cycles.

### 4.4.2 Duplicate-Free Enumeration Algorithm

We now give our algorithm for duplicate-free enumeration of assignments. As we shall show experimentally in Section 4.6, this algorithm is a significant improvement over the naive enumeration algorithm. In Section 4.4.3, we show how to make the algorithm adaptive, by taking user constraints into account. This will further reduce the size of the space that actually needs to be explored.

The main idea behind the algorithm, as alluded to earlier, is to avoid enumerating assignments that contain exactly \( k - 1 \) edges of a cycle of length \( k \) in the matching graph. Each such assignment is a duplicate of the assignment obtained by adding the remaining \( k \)-th edge in the cycle. We can formalize the “removal” of the assignments with \( k - 1 \) edges by imposing a set of constraints that the assignments must satisfy. Specifically, we use a Horn clause of the form \( x_1 \land x_2 \land \ldots \land x_{k-1} \rightarrow x_k \) to specify that every assignment that assigns 1 to \( x_1, x_2, \ldots, x_{k-1} \), \( x_{k-1} \) must also assign 1 to \( x_k \). We also take into account all “permutations” of such clauses, and we do this for all the cycles in the matching graph.

The net effect of this is that we reduce the problem of duplicate-free enumeration of assignments needed in our schema integration context to the problem of enumerating all the satisfying assignments for a set of Horn clauses. The latter problem has already been studied in
the literature, and there is a good algorithm for it. Indeed, Creignou and Hébrard [36] devised a polynomial-delay algorithm for generating all Boolean assignments that satisfy a set of Horn clauses. Polynomial-delay [58] means that the delay until the first satisfying assignment is generated, and thereafter the delay between the generation of any two consecutive satisfying assignments, is bounded by a polynomial in the input size (i.e., the total size of the input Horn clauses). This formalizes the notion of a tractable algorithm for an enumeration problem with a possibly exponential set of outputs.

Our resulting algorithm (CH-Enum) for duplicate-free enumeration of assignments (and the corresponding integrated concept graphs) is given in Figure 4.8. Note that, even though our algorithm uses a polynomial-delay algorithm as a subroutine, it is not itself a polynomial-delay algorithm, since, in the worst case, the number of cycles (and, hence, the number of Horn clauses) may be exponential in the size of the matching graph. Nonetheless, as detailed in Section 4.6, our algorithm performs well on both synthetic and real-life integration scenarios, and clearly outperforms naive enumeration.

We now briefly describe the Creignou & Hébrard procedure itself, as applied to our scenario. Let \( \Gamma \) be a set of Horn clauses as above, and let \( x_1, \ldots, x_n \) be the variables that occur in \( \Gamma \). Given such input, the algorithm proceeds by recursively considering the variables \( x_1, \ldots, x_n \) in order, as follows. Initially, all variables are unassigned. In step \( i \) (\( 1 \leq i \leq n \)), if the variable \( x_i \) is unassigned, then its value is set first to 1. Otherwise, the variable already has a value (see next), and the algorithm continues with step \( i + 1 \).

When \( x_i \) is set to 1, some of the Horn clauses in \( \Gamma \) may now have all 1’s in the left-hand side of the implication. For each such Horn clause, the algorithm tries to propagate the
**CH-Enum(G)**

**Input:** Matching graph \( G = (V, \text{HasA}, E) \)

**Output:** Enumeration of all the distinct integrated graphs (together with their mappings) that can be obtained from \( G \) via \text{ApplyAssignment}.

1. Compute all cycles of \( G \) (considering only the edges in \( E \)).

2. Construct a set \( \Gamma \) of Horn clauses as follows: for each cycle \( C = x_1, \ldots, x_k \) of edges in \( E \), add the following \( k \) Horn clauses to \( \Gamma \):
   
   \[
   x_1 \land x_2 \land \ldots \land x_{k-1} \rightarrow x_k
   
   x_2 \land x_3 \land \ldots \land x_k \rightarrow x_1
   
   \ldots
   
   x_k \land x_1 \land \ldots \land x_{k-2} \rightarrow x_{k-1}
   \]

3. [Creignou & Hébrard]: Generate all satisfying assignments \( X \) of \( \Gamma \).

4. For each assignment \( X \), output the result of \text{ApplyAssignment}(G,X).

---

**Figure 4.8: Duplicate-free enumeration algorithm.**

value 1 to the variable in the right-hand side of the implication, in an attempt to satisfy the Horn clause. If the variable on the right-hand side is not already assigned a value of 0 (from a previous step), then we set it to 1 (if not already 1) and continue to the next step \((i + 1)\). Otherwise, clearly, the clause cannot be satisfied with the current partial assignment. So, we abandon the branch with \( x_i = 1 \) and set \( x_i = 0 \). The algorithm continues with step \( i + 1 \). Whenever a full assignment is found, we output it and then backtrack to the last variable, \( x_i \), that was assigned 1. If no such \( x_i \) exists, we are done. Otherwise, we set that variable \( x_i \) to 0.
and proceed to explore, again, $x_{i+1}$.

The main advantage of this algorithm (over an exhaustive enumeration of satisfying assignments) is that as soon as a partial assignment is discovered to be unsatisfiable, none of its supersets are further considered. Hence, the algorithm prunes, early in the search, all the assignments that are guaranteed not to satisfy $\Gamma$.

### 4.4.3 Adaptive Enumeration

Our system does not enumerate all integrated schemas at once. Instead, target schemas are output one by one and the user is allowed to browse through the schemas generated so far, as well as request the generation of a new schema. As a result of this interaction, after examining a few integrated schemas, a user may gain more insight on the structure of the desired final schema. For example, the user may see things that are “wrong” and should be corrected. Based on this insight, the user can express additional constraints on how concepts should be merged. These constraints are then used to filter the set of schemas generated so far, and, more interestingly, can be incorporated in the enumeration of the subsequent schemas. The result is an adaptive enumeration procedure that significantly reduces the search space with every constraint learned from the user.

User constraints can be given through the visual interface of our system, directly in terms of concepts or of the matching edges between them. In contrast to the redundancy constraints described in Section 4.2.1 which are applied to merge $\text{HasA}$ edges between concepts, the constraints defined in this section express conditions on how to merge the concepts themselves. Furthermore, these constraints directly affect the enumeration procedure. Thus, we shall call
them *enumeration constraints*.

Our system allows two types of enumeration constraints. First, the user can require a matching edge $x$ to be always applied or never applied, expressed as $\text{Apply}(x)$ and respectively, $\neg\text{Apply}(x)$. Whenever such constraint is applied, the space of possible assignments is reduced in half, since one variable is eliminated (set to either 1 or 0). Second, the user can require constraints of the form $\text{Merge}(A_1, \ldots, A_n)$ or $\neg\text{Merge}(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are arbitrary concepts in the matching graph (some could be from the same schema). These constraints also reduce, significantly, the space of possible assignments.

The meaning of $\neg\text{Merge}(A_1, \ldots, A_n)$ is that, for every pair of distinct concepts $A_i$ and $A_j$, $i, j \in [1, n]$, $A_i$ and $A_j$ should never be merged. Thus, each concept $A_i$ will be in a different connected component. Constraints of the form $\neg\text{Merge}(A_1, \ldots, A_n)$ are useful for enforcing certain structural patterns that may be desired in the integrated schema. As an example, the user can add the constraint $\neg\text{Merge}(\text{org, location, emp, phone, fund})$ which essentially requires the structure of the second schema $S_2$ to be preserved. There is no merging among the five concepts, and their relative structure will stay the same in the integrated schema. However, the concepts of the first schema can be freely merged into the concepts of the second schema, based on the matching edges.

The semantics of $\text{Merge}(A_1, \ldots, A_n)$ is slightly more complex. This is due to the fact that it may not be possible to merge all the $n$ concepts, simply because there may not be enough matching edges. For example, $\text{Merge}($dept, fund$)$ cannot result in any merging, since there is no matching edge between dept and fund. The semantics we take is that we group the $n$ concepts into connected components based on all the matching edges in the matching graph.
If two concepts are in different components, we do not attempt to merge them. However, if two concepts are in the same connected component, we shall only enumerate integrated graphs that keep them connected. For example, the effect of \texttt{Merge(dept, org, grant, fund, project)} is that the subsequent enumeration shall only consider assignments that merge dept and org, and merge grant, fund, and project.

When a new enumeration constraint \( F \) is specified, the already generated schemas that do not satisfy \( F \) are filtered out. Conceptually, the filtering can be done by scanning all schemas which have been already shown to the user. In the system however, we postpone the work until the user wishes to examine one of the previously generated schemas. At that point, we test if the schema satisfies all enumeration constraints that have been added since the schema has been generated. More interestingly, we have modified the CH-Enum algorithm so that it generates, from that point on, only schemas that satisfy \( F \) (and the other existing enumeration constraints). The rest of this section describes this modification to CH-Enum. The main idea is to encode the enumeration constraints, whenever possible, with Horn clauses (similar to how we encode cycles in CH-Enum).

If \( F \) is of type \texttt{Apply}(x), where \( x \) is a matching edge, we add the Horn clause \( \rightarrow x \) to the set of clauses that are considered by CH-Enum, to encode the fact that every assignment must satisfy \( x = 1 \). In the implementation, if \( x \) is unassigned, then we fix \( x = 1 \) and continue the enumeration procedure. Here, \textit{fixing} the value of \( x \) means that \( x \) will never be subsequently changed during the enumeration procedure. If \( x \) is currently assigned 1, then we proceed with the enumeration until we reach the point when \( x \) has to be switched from 1 to 0. At this point, we fix \( x = 1 \) and backtrack to the variable before \( x \). If \( x \) is assigned 0, then we fix \( x = 1 \)
and backtrack to the variable before $x$. Hence, we eliminate any assignment that has $x = 0$. Similarly, if $\mathcal{F}$ is of type $\neg \text{Apply}(x)$, we add the Horn clause $\rightarrow \neg x$ to encode the fact that every assignment must satisfy $x = 0$. The implementation is similar (but complementary) to the case of $\text{Apply}(x)$.

If $\mathcal{F}$ is of type $\neg \text{Merge}(A_1, \ldots, A_n)$, then for every two distinct concepts $A_i$ and $A_j$, $i, j \in [1, n]$ we enforce the constraint $\neg \text{Merge}(A_i, A_j)$ as follows. For each simple path $x_1, \ldots, x_k$ between $A_i$ and $A_j$ in the matching graph, we use Horn clauses of the form $x_1 \land \ldots \land x_{k-1} \rightarrow \neg x_k$ (together with all the permutations). This encodes the fact that every assignment that assigns 1 to any $k - 1$ variables (edges) along the path $x_1, \ldots, x_k$ must assign 0 to the $k$th variable. Thus, we make sure that no path of edges connects $A_i$ and $A_j$.

Finally, the case of $\text{Merge}(A_1, \ldots, A_n)$ is trickier. There is no apparent way to encode such constraint, in general, as a set of Horn clauses. Hence, in the implementation, we cannot direct the CH-Enum algorithm to avoid the generation of violating assignments. However, once an assignment is generated, we can check whether it violates the constraint and then discard any such assignment. Although Merge constraints are not used to reduce the space explored by CH-Enum, they are still useful from the user interaction point of view.

### 4.5 System Architecture

We have implemented our schema integration methodology with Java 1.5 on top of the Clio system. The architecture of the system, illustrated in Figure 4.9, consists of four main modules. Sections 4.1.2 and 4.1.3 describe the methods underlying the module for extracting
the concept graphs from the source schemas, and respectively, the module for matching concepts. The ApplyAssignment and CH-Enum algorithms underlying the module for enumerating alternative integrated schemas have been described in Sections 4.2 and 4.4. The method for adaptively incorporating user constraints during enumeration is described in Section 4.4.3.

Figures 4.10 and 4.11 illustrate snapshots from the visual interface of the system, which plays an essential role in facilitating the designer’s understanding of the integration scenario. The visual interface displays multiple, correlated views:

- The source schemas and correspondences between their attributes (as in Figure 4.2);
- The source concept graphs (as in Figure 4.3);
- A side-by-side view of the matching graph and the integrated graph currently being ana-
Figure 4.10: Snapshot from the visual interface of the schema integration system: (a) the matching graph of source concepts; (b) an integrated concepts graph; (c) a corresponding integrated schema.

- A corresponding integrated schema (Figure 4.10(c));
- The mappings between the source and integrated schemas (Figure 4.11).

All views are synchronized in such a way that related entities across multiple views are highlighted when selected on any one view. For example, selecting the `dept-org` integrated concept in Figure 4.10(b) has several effects. First, the contributing source concepts, together
with the corresponding matching edges are highlighted, as illustrated in Figure 4.10(a). The set

types in the source schemas that correspond to these concepts, and the set type in the integrated

schema that corresponds to dept-org (Figure 4.10(c)) are also highlighted. Similarly, selecting

an attribute allows a user to visualize all “related” attributes in the schemas and concept graphs.

To facilitate the visualization of matching concepts, these concepts and their corre-

sponding integrated concept are aligned to appear closer to each other (as depicted in Fig-

dures 4.10(a–b)). Furthermore, the appearance of various elements is customizable. The user

Figure 4.11: Snapshot from the visual interface of the schema integration system: one of the

mappings between the sources and the integrated schema.
can resize, minimize or maximize concepts, and collapse entire groups of concepts in order to eliminate clutter, and be able to concentrate on specific parts of the integration problem.

4.6 Experimental Evaluation

We have evaluated the performance of our integration method on synthetic schema integration scenarios, as well as on real world scenarios. The experimental results show that CH-Enum is much faster compared to the naive enumeration and that it scales well with the increasing complexity of schemas and mappings. In particular, the average time to output the next integrated schema, when using CH-Enum, is fairly low. Furthermore, experiments with enumeration constraints indicate that the space of candidate schemas drastically reduces after adding even a small number of constraints. All experiments are performed on a PC-compatible machine, with two 2.0GHz P4 CPUs and 4Gb RAM, running Linux and JRE 5.0.
4.6.1 The Synthetic Scenarios

The goal of the experiments with synthetic scenarios is to measure the performance of our integration method along three dimensions: 1) the number $N$ of input schemas, 2) the complexity of each schema, measured by its root fanout $F$ (i.e., the number of top level sets) and nesting depth $D$ (i.e., the number of levels of nested sets), and 3) the degree $I$ of interconnection between the input schemas. Figure 4.12 shows $N = n$ synthetic schemas with root fanout $F = 1$ and nesting depth $D = d$, and a mapping that connects $S_1$ and $S_2$. Given input schemas $S_i$ and $S_j$, either $S_i$ and $S_j$ are not connected, or they are connected through a set of correspondences $V_{ij}$ relating each attribute $B_k$ of $S_i$ with the attribute $B_k$ of $S_j$. For $F = f$, the configuration shown in the figure repeats $f$ times (since there are $f$ top-level sets in each schema). The degree $I$ of interconnection between input schemas is the number of schemas $S_j$ that are connected to an input schema $S_i$, averaged over all input schemas.

Experiment 1 The goal of this experiment is to compare the performances of the naive enumeration and CH-Enum algorithms. In this experiment the degree of interconnection was fixed to $I = 3$ (i.e., each input schema relates on average to three other schemas). The complexity of each schema was fixed to nesting depth $D = 3$ and root fanout $F = 1$, and the number $N$ of input schemas was varied from 4 to 10. Figure 4.13 reports the average time to generate the next integrated schema, for both strategies. (In each scenario and for each strategy, we stopped after generating the first 1000 integrated schemas.)

The results show that CH-Enum performs much better than the naive enumeration strategy. For example, in the scenario with $N = 8$ CH-Enum took 2 milliseconds, on av-
average, while the naive algorithm took around 370 milliseconds, to output the next integrated schema. The difference in performance is not unexpected, since the naive algorithm exhaustively enumerates all possible assignments, a large portion of which are duplicate assignments (i.e., leading to the same set of connected components in the matching graph). These duplicates are explicitly removed by checking whether the set of connected components generated in Step 1 of ApplyAssignment has already been encountered. The space needed to store the previously seen sets of connected components is in itself a problem. (In fact, for the scenarios with \( N = 9 \) and \( N = 10 \), the naive algorithm ran out of memory, which is why the times are not reported in Figure 4.13(a).) To illustrate the savings obtained with CH-Enum, consider the scenario with \( N = 4 \), where there are 3375 distinct integrated schemas. In this scenario, the matching graph consists of 12 concepts and 18 matching edges, and has 21 cycles. The naive enumeration strategy performs duplicate elimination on \( 2^{18} \) possible sets of connected components, while CH-Enum directly generates the 3375 distinct ones.

For the rest of the experiments, we report only the times obtained with CH-Enum, since it outperforms the naive enumeration strategy.

**Experiment 2** The goal of this experiment is to test the scalability of CH-Enum with the complexity of the input schemas. Implicitly, this also tests the scalability of the ApplyAssignment algorithm, which is invoked to generate each integrated schema. We used scenarios with two input schemas (i.e., we fixed \( N = 2 \) and \( I = 1 \)) and generated the first 1000 integrated schemas. Figure 4.13(b) illustrates the influence of the root fanout \( (F) \) and the nesting depth \( (D) \) of the input schemas on the average time to output the next integrated schema. As expected, the performance decreases with the increase in the complexity of the schemas.
4.6.2 The Real Scenarios

We tested the performance and usability of our method in several real-life integration scenarios. The schemas used in these scenarios are: a relational and an XML schema, each representing gene expression experimental results (GENEX); a fragment of the Genomics Unified Schema (GUS) [54] and the BioSQL schema for genomic sequences and features [25]; two XML schemas representing enterprise business objects related to orders, one from SAP and
the other one in use with the IBM WebSphere Business Integration (WBI) suite; a relational and the DTD version of the Mondial database [68]; two relational schemas from the Amal-
gam integration benchmark [69] for bibliographic data; two variations of the XML schema for the DBLP bibliography; the first schema in the Amalgam benchmark and one of the previous DBLP schemas; and three XML schemas, each with a different nesting structure, represent-
ing information about departments, projects and employees. Figure 4.14 shows, for each case, the number of schemas, as well as the number of concepts, matching edges and cycles in the matching graph.
<table>
<thead>
<tr>
<th>Integration Scenario</th>
<th>Enumeration Constraints</th>
</tr>
</thead>
</table>
| GUS-BioSQL           | Merge(BioEntry, GOTerm, GOSynonym, Gene, GeneSynonym, Term, TermSynonym)  
|                      | Apply(GORelationship ← TermRelationship)  
|                      | Apply(Taxon₁ ← Taxon₂)  
|                      | Apply(TaxonName₁ ← TaxonName₂) |
| WBI-SAP              | ¬Apply(Address ← SAP_Order)  
|                      | Merge(Order, PaymentInformation, SAP_Order, SAP_OrderDateData)  
|                      | Apply(OrderLineItem ← SAP_OrderLineItem)  
|                      | Apply(Adjustment ← SAP_OrderLinePricing)  
|                      | Apply(DeliverySchedule ← SAP_ScheduleLines) |
| Amalgam-DBLP         | ¬Merge(masterthesis₁, phdthesis₁, author₁, techreport₁, book₁)  
|                      | Merge(author₁, author₂, author₃, author₄)  
|                      | Apply(article₁ ← article₂)  
|                      | Apply(inproceedings₁ ← inproceedings₂)  
|                      | Apply(book₁ ← book₂) |
| Proj-Dept-Emp        | ¬Apply(dept₁ ← project₂)  
|                      | ¬Apply(project₂ ← dept₃)  
|                      | Merge(emp₁, emp₂, emp₃)  
|                      | Apply(dependent₂ ← dependent₁)  
|                      | Merge(project₁, project₂, project₃) |

Table 4.1: Some of the enumeration constraints used in experiments with real schema integration scenarios.

The running times for CH-Enum, as well as the size of the space of candidate schemas in each scenario are shown in Figure 4.15. CH-Enum performed well in all cases, taking up to 35 milliseconds to generate the next integrated schema, on average. The size of the space of candidate schemas (shown in the Integrated Schemas column) may be large (around or above 1000 schemas in the cases of Mondial, Amalgam, DBLP, Amalgam-DBLP). However, our experiments with user constraints, described next, show that the number of schemas that a user actually explores (and the system generates) before arriving at the desired integrated schema is
much smaller.

The last three columns in Figure 4.15 describe the user interaction effort in terms of the number of enumeration and redundancy constraints that need to be added. In the case of enumeration constraints, we show the impact that such constraints have on the overall convergence of our schema integration method. Specifically, the space of remaining candidate schemas (i.e., the space of schemas that satisfy the enumeration constraints added so far and, hence, are of interest to the user) is significantly pruned after adding just a few enumeration constraints. In the DBLP scenario, for example, the number of schemas of interest is decreased from 1096 to 144 after adding the first constraint, then further decreased to 4 after the third constraint, and then decreased to just one after adding the fourth constraint. In the figure, these facts are denoted as $1^{st} \Rightarrow 144; 3^{rd} \Rightarrow 4; 4^{th} \Rightarrow 1$.

To illustrate the enumeration constraints that were added, the following are the four enumeration constraints for the DBLP scenario: 1) do not merge concepts in the second DBLP schema (thus, the second DBLP schema is taken as a reference schema and the other schema is merged into it), 2) merge Article with Pub, 3) merge Inproceedings with Pub, and 4) merge Author (in the first schema) with Author (in the second schema). For Mondial, we used enumeration constraints to enforce the merging of matching concepts such as Sea, Mountain, Lake and others, occurring in the two schemas. As another example, just four enumeration constraints sufficed to reduce the space of candidate schemas to only two integrated schemas of interest in the WBI-SAP scenario. The enumeration constraints we have added in the WBI-SAP scenario, as well as in some of the other scenarios, are shown in Table 4.1.

We further note that the number of enumeration constraints required to prune the
search space to a few schemas of interest can be seen as a pessimistic upper bound on the number of enumeration constraints that the user has to actually enforce in practice. This is because the system may generate and display the “right” integrated schema much earlier in the process. The reason for this behavior is that the implementation of the enumeration algorithm gives priority to the most merged candidate schemas, which are often what a user wants, provided that other constraints are satisfied. Thus generating schemas from the most merged ones to the least merged ones (due to the fact that we explore assignments starting from all 1’s down to all 0’s) is quite beneficial in practice.

To illustrate this behavior, the second to last column in Figure 4.15 shows the number of constraints we have actually enforced before our system displays the right integrated schema, in each scenario. In the WBI-SAP scenario, for example, the desired integrated schema was generated and displayed after we added the first enumeration constraint, although 5 such constraints are (theoretically) necessary to reduce the search space to this integrated schema. In this scenario, as well as in several others (e.g., Genex, GUS-BioSQL, DBLP), most or all of the desired merging is performed by our system automatically, and the user needs to enforce at most one constraint to indicate, through \texttt{Apply} or \texttt{Merge} constraints, which merging choices are undesirable.

Furthermore, as shown in the last column, at most 12 redundancy constraints were needed, over all scenarios, to remove the redundant \texttt{HasA} edges in the integrated schema. As an example, in the DBLP scenario, only one redundancy constraint was needed to merge two parallel relationships “copied” from Author \texttt{HasA} Article and respectively, Author \texttt{HasA} Inproceedings. Finally, we note that in the case of GUS-BioSQL, all that is needed is 4 enumera-
tion constraints (at most) and 6 redundancy constraints to arrive at the merge of two complex schemas of real-life significance to biologists. A similar comment applies to the WBI-SAP scenario, where the input schemas are also quite complex (the concepts have tens of attributes) and are of real-life significance to enterprise applications.

As a note on the implementation of our method, we observe that the number of cycles in the matching graph impacts the initialization time of CH-Enum (i.e., the time to compute all cycles in Step 1). In the Amalgam scenario (our worst case), it takes 92 seconds to compute a total of 3,486 cycles. In general, a large number of cycles in the matching graph constitutes a potential problem for Step 1 of CH-Enum. In the implementation, we can use a “hybrid” approach between the naive enumeration and CH-Enum that limits the number of cycles computed in Step 1, and checks for duplicate assignments as an extra step (since duplicate assignments may still appear due to cycles not found in Step 1). Thus, we bound the initialization time, at the expense of an increase in the time to generate the next schema.

4.7 Related Work

The distinguishing feature of our approach when compared to existing work on schema integration, model merging, and ontology merging, is the systematic enumeration and exploration of the different integration designs. The enumeration of alternative designs is based on the recognition that correspondences between concepts signify overlap in semantics rather than equivalence, and therefore such concepts may or may not be merged, depending on the scenario. Beyond enumeration, there are several other differences and similarities with the existing work
that are worth noting. We focus our discussion on the model merging method of Pottinger and Bernstein [78], since this subsumes much of the earlier work on schema integration [14, 26, 87] and also includes merging-specific features that are present in PROMPT [72] and other ontology merging systems, such as FCA-Merge [88]. We note, in this context, that most ontology integration literature has been primarily focused on the problem of ontology alignment, which is deriving relationships across concepts in different ontologies (see ILIADS [94], for a recent example). In contrast, our focus here is on exploring the alternatives for the structural unification of the redundant concepts or attributes (i.e., the merge phase).

One of the main features in [78] is the use of a mapping “in the middle” that essentially drives the integration of the two input models. The mapping, which can be quite complex, can be seen as a “template” for the integrated model, and must be specified by a user before the actual integration. In contrast, the input to our method is just a set of atomic correspondences which can be discovered by an automatic schema matching system. The additional user constraints in our method are given as the user explores the available choices that the system discovers (a “learn-as-you-go” approach, as opposed to knowing or guessing in advance what the outcome should be). As in [78], our method operates at a logical rather than physical schema level. Our meta-meta-model (using their terminology) is more basic; it includes HasA edges as the basic form of relationships, and a simpler form of Contains (i.e., concepts contain attributes). Nevertheless, our graphs of concepts can express most of the essential features that appear in schemas or in conceptual models such as ER diagrams or UML class diagrams. The work of [78] is extended in [79] by considering the schema integration problem in the context of source schemas related by GLAV mappings, as opposed to just correspondences between
schema attributes. It would be interesting to investigate how our enumeration methodology extends to this context. Finally, the work of [78] addresses the treatment of type and representation (e.g., name) conflicts. The resolution of such conflicts can be applied as a post-processing step, and is complementary to our basic schema integration method.
Chapter 5

Conclusions

This dissertation describes methods, and corresponding system implementations, for enabling interoperability of heterogeneous databases through provenance and schema integration. Provenance facilitates a designer’s understanding of the quality of the data, as well as the quality of the mappings used in applications requiring data interoperability. In the DBNotes and SPIDER systems, we have studied the problem of tracing data provenance for two common mapping formalisms: SQL queries, and respectively, schema mappings. In doing so, we have investigated both the eager approach (in DBNotes), and respectively, the lazy approach (in SPIDER) for computing provenance, each useful in different contexts. Schema integration enables data interoperability in that it provides a standard representation for heterogeneous data sources. The schema integration system that we have developed facilitates the process of designing an integrated schema via an adaptive and interactive method for exploring the space of all candidate schemas suitable for the integration application at hand. Furthermore, the system generates information-preserving mappings between the sources and the integrated schema. Our methods
for tracing data provenance can be subsequently applied to further enhance a user’s understanding of the auto-generated mappings.

The rest of this chapter summarizes several interesting directions for future research.

**Other Mapping Formalisms** It would be interesting to investigate whether our approaches for computing provenance can be extended to handle more expressive mapping formalisms. While the default propagation scheme of DBNotes can be easily extended to handle Select-Project-Join-Union queries with aggregates, inequalities, or bag semantics, an interesting open question, which involves reasoning about equivalence of queries within these classes, is whether the default-all behavior can be simulated with finitely many queries for such extensions. Another interesting direction is to extend our notion of routes to handle egds, as well as the recent nested schema mapping language of [47]. Here, the challenge is to explain the data merging that occurs in the target instance due to the application of an egd, or due to the grouping semantics of nested mappings [47].

**Algorithmic Issues** An interesting future direction in DBNotes is to investigate optimization opportunities for the query basis, and improve the performance of the default-all propagation scheme. In SPIDER, we have shown that our ComputeOneRoute algorithm outputs a route in polynomial time. Furthermore, we have modified this algorithm to output alternative routes, as needed. However, it remains an interesting open question whether alternative routes are output with polynomial-time delay. Although our experiments with the schema integration system show that the overhead of computing the cycles in the matching graph is manageable in realistic scenarios, an interesting open question is whether there exists a polynomial-time delay algorithm for enumerating non-duplicate assignments.
**Schema and Mapping Evolution** In data interoperability scenarios, changes naturally occur in practice. New sources may join, while others may leave the system. Furthermore, schemas and mappings between schemas typically evolve over time. In debugging with SPIDER, for example, the user may change the schema mapping repeatedly, in order to arrive at the desired specification. As a consequence, the target instance may no longer satisfy the new schema mapping and in practice, it may be wasteful to recompute the target instance from scratch every time a change occurs in the schema mapping. It would be interesting to study the problem of incrementally adapting the target instance to satisfy the new schema mapping, dynamically, along with the changes in the schema mapping. This is related to the view adaptation problem, where the goal is to incrementally update a materialized view in response to changes in the view definition. Techniques for incrementally adapting the target instance when the mappings evolve will require reasoning about provenance, in order to identify the affected data. The objective is to develop techniques that can be easily deployed on existing mapping systems such as Clio, or RDA. Hence, SPIDER’s lazy approach for computing provenance is preferable to the eager approach of DBNotes, since the latter requires changes to the underlying transformation engine.

We are currently extending our schema integration system in order to support incremental adaptation of the integrated schema and associated mappings with changes in the original integration context. Preliminary investigations indicate that valuable human effort can be saved by incrementally adapting the outcome of a previous integration, as opposed to restarting the design process from scratch, each time the context evolves.

**Usability Issues** Designing data interoperability applications is a human-involved process in general. Hence, usability is an important factor. We believe there are several directions for
improving the usability of our systems. First, a mapping designer must manually modify a mapping that has been identified as incorrect in the process of understanding and debugging mappings with DBNotes and SPIDER. To reduce the amount of human intervention required, it would be interesting to develop techniques for automatically suggesting how mappings should be adjusted or “repaired”, possibly based on hints that the designer can easily specify.

Second, the size of the route forest computed with SPIDER may be large, hence overwhelming for a designer. It would be interesting to explore other techniques for assisting the designer in exploring the route forest, in addition to the “user-guided computation of routes” feature currently offered by SPIDER. There are scenarios in which the route forest embeds many “similar” routes. Such similar routes consist of sequences of satisfaction steps which use the same tgds, but different homomorphisms. It would be interesting to develop summarization techniques for the route forest that group similar routes together, thus allowing one to focus on a smaller number of routes. A naive approach for computing such a summary of the route forest is to first compute all routes, then group the similar routes. The challenge is to develop more efficient algorithms which avoid the exploration of multiple “similar” routes, if possible.

Third, our schema integration methodology relies on enumeration constraints in narrowing down the space of candidate schemas. It would be interesting to investigate techniques for using source data to illustrate and suggest possible enumeration constraints to the designer. Furthermore, it would be interesting to develop other methods for exploring the space of candidate integrated schemas, besides enumeration based on user constraints. A natural direction is to develop techniques for ranking integrated schemas, based on their adequacy given an expected query workload, or by considering weights or probabilities on the matchings between
attributes and concepts. Such techniques might improve the usability of the system by leveraging any available extra information (e.g., a query workload) to reduce the number of schemas presented to the user.

Finally, it would be interesting to systematically evaluate the usefulness of our techniques in providing developmental support to designers of data interoperability applications, by means of a formal human-computer interaction study.
Bibliography


239


[70] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in


[100] C. Yu and L. Popa. Constraint-Based XML Query Rewriting For Data Integration. In