Oblivious Queries over Data with Irregular Structure

Thesis Submitted for the Degree
“Doctor of Philosophy”
by

Yaron Kanza

Submitted to the Senate of the Hebrew University
October 2004
This work was carried out under the supervision of:

Prof. Yehoshua Sagiv
Acknowledgments

First and foremost, I am deeply grateful to my advisor, Shuky Sagiv, for believing in me (more than I believed in myself), and for encouraging me to choose the academic path. Shuky, thank you for the research environment that you provided to me during the last years. You have taught me to aspire in research and to dare when teaching; to fear not testing new ideas and exploring new directions. I greatly appreciate that you always had time for me, even when you had so many other duties. I thank you for your generous and continues support, and also for your excellent advice throughout the years—in research and for life. It has been a great pleasure being your student.

I wish to thank my colleagues in the database group, and especially my collaborators: Sara Cohen, Benny Kimelfeld and Eli Safra. It has been a great fun working with you. I also want to thank Catriel Beeri, for many interesting and useful discussions on various database topics, and for providing, in many occasions, valuable insight on my work. I wish to thank Werner Nutt who helped me when I made my first steps in research.

Special thanks I owe to my parents, Bilha and Eli, for their endless help in so many things. Mom, Dad—thank you, it would have been extremely difficult for me to complete this thesis on time without your help with Stav. I also wish to thank Michal’s parent, Yoram and Neomi, for helping me and Michal in the months when I had to dedicate most of my time to writing this thesis.

Finally, I wish to thank my wife, Michal, and my daughter, Stav, for their willing to carry the burden of a Dad that is busy with research. Michal, Stav—at the end of the day, it is all for you.
Abstract

In traditional query languages, formulating queries requires an exact knowledge of the schema. In particular, the user should explicitly take into account the possibility of incomplete information. Oblivious querying is an alternative approach that alleviates the need for an exact knowledge of the schema. Specifically, oblivious querying was devised for handling the following cases. First, when data have an irregular structure (e.g., data collected from heterogeneous sources). Second, when the schema of the data is complicated or is frequently changed. Third, when there is no schema or the schema is unknown.

Our approach to oblivious querying of semistructured data is twofold and is based on the paradigm of flexible queries and the paradigm of maximal answers. The paradigm of flexible queries facilitates, in a novel way, easy and concise querying that is independent of any particular representation of the relationships among entities. Two different semantics, flexible and semiflexible, that admit different levels of flexibility are introduced and investigated. The complexity of evaluating queries under the two semantics is analyzed. Query evaluation is polynomial in the size of the query, the database and the result in the following two cases. First, a semiflexible DAG query and a tree database. Second, a flexible tree query and an arbitrary database.

Query containment and equivalence are also investigated, as a first step towards developing optimization techniques. For both the semiflexible and flexible semantics, query containment and equivalence can be tested in polynomial time.

Under the semiflexible and flexible semantics, two databases could be equivalent even when they are not isomorphic. Database equivalence is formally defined and characterized. Moreover, the complexity of deciding equivalences among databases is analyzed. The implications of database equivalence on query evaluation are explained. In particular, transforming a database graph into an equivalent tree is investigated for both the semiflexible and the flexible semantics. Transforming a database to an equivalent tree is important for representing the data as an XML
document with no references. *Tree transformable* databases are characterized and a transformation algorithm is provided.

The second paradigm, in our approach, is of queries with maximal answers. This paradigm is based on the notion of answers that are partial bindings of query variables to database objects. It facilitates a formulation of queries that is independent of whether the data is incomplete or not. The answer to a query consists of maximal rather than complete matchings, i.e., some query variables may be assigned null values. Answers that can be extended, by mapping more variables, are discarded since they are not maximal. In the relational model, a similar effect is achieved by computing the *full disjunction* (rather than the natural join or equijoin) of the given relations. For two semantics that allow maximal answers, the or-semantics and the weak semantics, it is shown that query evaluation has a polynomial-time complexity in the size of the query, the database and the result.

This work shows how to combine the two paradigms and their proposed semantics. For combined semantics, query evaluation is polynomial in the size of the query, the database and the result in the following two cases. First, a tree query under the combination of the semiflexible semantics with either the weak or the or-semantics. Secondly, a query and a database that are arbitrary graphs, under the combination of the flexible semantics with either the weak or the or-semantics.

In the last part of the work, it is shown that the evaluation of full disjunctions is reducible to query evaluation under the weak semantics and hence can be done in polynomial time in the size of the input and the output. Complexity results are also given for the problem of evaluating a projection of the full disjunction. In the special case of $\gamma$-acyclic relation schemes, the projection problem has a polynomial-time algorithm in the size of the input and the output. In the general case, such an algorithm does not exist, assuming that $P \neq NP$. Finally, it is shown how to use full disjunction for transforming a semistructured database into a universal relation. This transformation admits querying a single relation with a known schema as an alternative to querying a graph whose schema may be unknown.
Contents

Acknowledgement

1 Introduction 1

2 The Data Model 10
  2.1 Databases .................................................. 10
  2.2 Queries .................................................... 11

3 Flexible and Semiflexible Queries 14
  3.1 Flexible and Semiflexible Matchings ....................... 15
    3.1.1 Examples ............................................. 18
    3.1.2 More on Semiflexible Matchings ..................... 20
  3.2 Query Evaluation in Semiflexible Semantics ............. 30
    3.2.1 Path Queries ......................................... 31
    3.2.2 Tree and DAG Queries over a Tree Database .......... 34
    3.2.3 Cyclic Databases and Cyclic Queries ............... 59
  3.3 Evaluating Flexible Queries ............................. 60
  3.4 Query Containment and Query Equivalence ............... 63
  3.5 Database Equivalence .................................... 66
    3.5.1 Removing Redundancies .............................. 70
    3.5.2 Transforming a Database to an Equivalent Tree ...... 72
  3.6 Summary of Contributions ................................ 84

4 Queries with Maximal Answers 86
4.1 Query Semantics ............................................. 86
  4.1.1 Partial Matchings ..................................... 86
  4.1.2 Subsumption and Maximal Matchings ............... 89
4.2 Computing Maximal Matchings ............................. 91
  4.2.1 The Product Graph .................................. 91
  4.2.2 Computing Maximal OR-Matchings .................... 93
  4.2.3 Computing Maximal Weak Matchings ................. 100
4.3 Summary of Contributions ............................... 101

5 Combining Flexibility with Maximal Answers ............. 102
  5.1 Queries under the Combined Semantics ................. 102
    5.1.1 Combined Semantics ................................ 102
    5.1.2 Examples ......................................... 105
  5.2 Computing Maximal Matchings .......................... 107
    5.2.1 Computing Semiflexible-OR-Matchings ............... 107
    5.2.2 Computing Flexible-OR Matchings and Flexible-Weak Matchings ............................................. 118
  5.3 Summary of Contributions ............................. 120

6 Full Disjunctions ............................................. 121
  6.1 Motivation ............................................. 122
  6.2 Preliminaries .......................................... 124
  6.3 Transforming Semistructured Data to Relational ....... 125
  6.4 Computing Full Disjunctions ............................ 127
  6.5 Projections and Restrictions of Full Disjunctions ...... 131
  6.6 Generalizing Full Disjunctions .......................... 133
  6.7 Summary of Contributions ............................. 134

7 Conclusion .................................................. 136

Bibliography .................................................. 138
Chapter 1

Introduction

Querying Semistructured Data and XML

Semistructured data occur in situations where information has an irregular structure and is incomplete. Consequently, the semistructured data model is aimed at representing data that do not conform to a strict schema. The lack of a schema may be the result of frequent changes in the structure of the data. It can also be typical of an environment where many different users contribute data to the database, in a variety of forms. The World-Wide Web is such an environment [1, 2, 13].

The semistructured data model is formally described in Chapter 2. In the semistructured data model, databases are rooted labeled directed graphs. Nodes in the graph represent objects. Relationships between pairs of objects are represented by labeled edges. Values are attached to atomic nodes, i.e., nodes that have no outgoing edge. In many cases, the data is self-describing. That is, there is no schema that explicitly defines the structure of the data. Instead, the structure of the data is implicitly described by the graph.

Recently, the eXtensible Markup Language (XML) [64] has become a popular notation for semistructured data. Essentially, XML has been developed for data exchange between applications. In data exchange, the sender and receiver do not necessarily use the same schema to represent data. Hence, XML documents do not have to conform to a schema. Moreover, schemas for XML (either DTD or XML Schema [63]) do not rigorously define the structure of conforming documents. Thus, in many cases, even XML data that conform to a schema are incomplete or do not
comply with a strict pattern.

Traditional query languages and traditional querying methods are not well suited for semistructured data or XML. For one, the semistructured data model is based on a labeled directed graph. More importantly, traditional query languages are not adapted to data having no schema at all, nor to data whose schema may change considerably over time or from one data instance to another. An additional consideration is the size of the schema, which could be quite large compared to the size of schemas for structured data.

When the schema is large and complicated, querying the data could be rather difficult. An initial phase of querying the schema might be needed before the query can be formulated. Even with this additional step, the query could be quite large and hard to phrase, due to the need to cover many structurally similar, but not structurally identical data instances. Furthermore, an increase in the size of the query may cause an increase of the time complexity.

The size of the schema is not the only source of difficulties. A case in point is an XML repository of documents with DTDs designed for machine interchange (as well as DTDs that were actually generated by machines). In the eyes of a layperson, such DTDs might not be easy to grasp, regardless of the size.

Traditional query languages are based on the concept of rigid matchings. In a rigid matching there should be a perfect match between the conditions specified in the query and the data. Rigid matchings are very sensitive to variations in the schema and to incompleteness of the data. A query that is posed having a specific schema in mind, is likely to yield only some of the answers or no answer at all, even when only minor changes occur in the structure of the data while the data itself does not change at all.

In recent years, a considerable amount of work has been done on querying semistructured data, in general, and XML data, in particular. Many query languages for semistructured data [4, 7, 8, 15, 46, 50] and for XML [2, 9, 17, 22, 30] have been proposed. In order to deal with the special characteristics of semistructured data, some of these languages use regular expressions. Issues related to the evaluation and optimization of query languages with regular expressions are discussed in
Chapter 1 Introduction

[6, 51]. Additional work has been done on constraining semistructured data [14, 16], describing the structure of the data for query formulation and optimization [36], and extracting information on changes in the data [18].

The query languages cited above are based on rigid matchings, and some of these languages use regular expressions to express possible variations in the structure of the data. But the approach of relying on regular expressions is problematic in at least two aspects. First, the ounce of responsibility is on the user, who should phrase the regular expressions in a way that will cover the possible variations in the structure of the data and yet will not cover too much. Secondly, the tasks of query evaluation and optimization become more complex in the presence of regular expressions.

An alternative approach is that of extracting the ontology from the schemas (e.g., DTDs). The ontology consists of the list of names given to the elements and to the attributes of the schemas, and is much easier to comprehend than the full structural details of the schemas. In many cases, queries can be phrased simply and succinctly using just the ontology.

Semiflexible and Flexible Queries

In Chapter 3, we present two semantics that are suitable for ontology-based querying of semistructured data. Similarly to existing query languages, our queries are also represented as rooted labeled directed graphs. Thus, preserving the similarity between queries and databases. However, instead of a semantics that is based on rigid matchings, we introduce two new semantics that are based on semiflexible matchings and flexible matchings. We believe that these semantics capture the intended meaning of many common queries. A user only needs to be familiar with the ontology of the database in order to phrase a query, and hence query formulation is more intuitive and simpler compared to query languages that use regular expressions.

In a rigid-matching semantics, the implicit relationships between objects are expressed by labeled edges. More complicated relationships have to be constructed
explicitly by formulating queries in terms of labeled directed graphs. The semiflexible semantics, in comparison, also uses labeled directed graphs to describe relationships between objects, but it assumes that database objects are implicitly related if they are connected by a directed path (and not just by an edge). Thus, a path $\pi$ of the query can match a path $\phi$ of the database if $\phi$ includes all the labels of $\pi$; however, the inclusion need not be contiguous or in the same order as in $\pi$. The flexible semantics further extends this idea by applying transitivity to the implicit relationships that hold under the semiflexible semantics.

In Chapter 3, we investigate query evaluation under the semiflexible and the flexible semantics. Other topics that are included in this chapter are the followings. First, we characterize query equivalence and query containment. Secondly, a novel concept of database equivalence is presented. We provide a characterization of equivalent databases, for both the semiflexible and the flexible semantics. Thirdly, it is shown that some databases could be transformed into equivalent trees. We characterize these databases and provide an algorithm that performs the transformation. The work that is presented in Chapter 3 has been published in [43]. A query language that can query simultaneously relations and XML documents and uses the flexible semantics has been presented in [23].

Queries with Maximal Answers

Queries under the rigid, the semiflexible and the flexible semantics contain variables and constraints on these variables. An answer to a query is a binding of all the query variables to database objects, according to the query constraints. However, when querying incomplete data, the binding of query variables should be relaxed to allow partial answers, i.e., answers in which some of the variables do not have a binding to a database object. Otherwise, the user might receive only some of the answers that she expected. Alternatively, the user should know where data might be incomplete and formulate the query accordingly. However, when the structure of the data is unknown or when the schema is complicated, it is practically impossible to know where data might be incomplete.
An important issue in queries with partial answers is subsumptions among answers. An answer is \textit{subsumed} by another answer if the subsuming answer has all the bindings of the subsumed answer plus some additional bindings. There is no need to keep subsumed answers in the result. For example, if the tuple \((a, b, c)\) is an answer, then there is no need to include also the tuple \((a, b, \perp)\) as an answer. If subsumption is not handled judiciously during query evaluation, then it could add an exponential factor to the size of the query result without adding new information. Thus, queries should return only \textit{maximal answers}, i.e., answers that are not subsumed by any other answer.

Several semantics for queries with maximal answers over semistructured data were proposed in [41, 42]. The semantics differ in their degree of incompleteness. Query formulation remains the same under all these semantics, but the user can choose the degree of incompleteness that suits her best.

A significant issue is the complexity of evaluating queries with incomplete answers. Traditionally, complexity of query evaluation is measured in terms of either data complexity or query complexity [61]. But neither one can capture the differences in time complexity between evaluation of queries under complete semantics versus the incomplete semantics of [41, 42]. For all these semantics, data complexity is polynomial and query complexity is exponential.

The move from the traditional complete semantics to an incomplete semantics could potentially have two opposing effects. On one hand there are more answers, but on the other hand finding some answer could be easier, since it is not necessary to satisfy all the conditions of the query.

A more suitable measure of complexity is \textit{input-output} complexity, in which the time to evaluate a query is measured as a function of the size of the input (i.e., both the query and the database) and the output. Yannakakis [65] used this complexity measure to show that there is a polynomial-time algorithm for evaluating the natural join of \(n\) relations if the relation schemes are \(\alpha\)-acyclic. In the general case, deciding non-emptiness of the natural join of \(n\) relations is NP-complete [49].

Two of the semantics, namely, the \textit{or-semantics} and the \textit{weak semantics} that were presented in [41, 42] are investigated in our work. In [41, 42], it was shown
that for both the OR-semantics and the weak semantics, tree as well as DAG queries (without selections and projections) can be evaluated in polynomial time in the size of the input and output. In Chapter 4, we extend this result to queries that may have cycles. The work that is presented in Chapter 4 has been published in [44].

Combining the Paradigms of Flexible Queries with the Paradigm of Maximal Answers

Thus far, we discussed two paradigms. The paradigm of flexible queries and the paradigm of maximal answers. The first paradigm has been developed for managing data that has an irregular structure or an unknown schema. The second paradigm handles incomplete information. As mentioned previously, semistructured data occur in situation where information has an irregular structure and is incomplete. Hence, when querying semistructured data, the two paradigms should be combined.

In Chapter 5, four new semantics are presented. Each new semantics is a combination of a semantics of the flexible-queries paradigm (i.e., the semiflexible and the flexible semantics) with a semantics of the maximal-answers paradigm (i.e., the weak and the OR-semantics). The new semantics facilitate the querying of data that is both irregular and incomplete. Query evaluation is investigated under the new semantics.

Full Disjunctions

Incomplete information may also occur in relational data. Usually, relational query language return complete answers, that is, answers that provide bindings to all the variables of the query. If the database has incomplete information, the user has to formulate the query differently, e.g., by using outerjoins or conditions that check whether some attributes are null. The need to be aware of the possibility of incomplete information is further complicated by the fact that the semantics of outerjoins is problematic, since outerjoins are not associative.

Galiando-Legaria [34] proposed full disjunctions, which are commutative and
associative, as an alternative to outerjoins. Essentially, the full disjunction of a set of relations is obtained by taking all joins of any subset of relations, excluding joins that involve a Cartesian product, and removing subsumed tuples. Rajaraman and Ullman [55] showed that the full disjunction of \( n \) relations can be computed by a sequence of natural outerjoins if and only if the relation schemes form a connected and \( \gamma \)-acyclic hypergraph. Rajaraman and Ullman also enunciated full disjunctions as a mechanism for integrating information from the Web, since the user cannot have any a priori knowledge of where information might be incomplete.

The size of a full disjunction could be large and sometimes even exponential in the size of the input. Thus, it is important that the computation of full disjunctions will be performed in polynomial time, under input-output complexity. For relations with relation schemas that form a connected and \( \gamma \)-acyclic hypergraph, a polynomial-time algorithm, under input-output complexity, has been proposed by Rajaraman and Ullman [55]. For the general case, it has been unknown whether a polynomial-time algorithm exists.

In Chapter 6, we show that computing full disjunctions is reducible to evaluating queries under the weak semantics. Thus, it follows that full disjunctions can always be computed in polynomial time in the size of the input and the output. We also give complexity results for two related problems. One is evaluation of a projection of the full disjunction. The other is evaluation of all tuples of the full disjunction that are not null on a given set of attributes.

We also propose a novel way, for oblivious querying of semistructured data, that is based on full disjunctions. This querying method can be applied to tree databases that do not have repeated labels in their paths. In this querying method, the semistructured database is transformed into a universal relation. Consequently, the user can query a relation with a known schema instead of querying the semistructured data. A similar approach has been used in [24, 25] for queries that are just selection and projection over XML. Our approach is different from the approach of [24, 25] in the following aspect. It finds connections between objects based on paths from the root to the leaves of the database, as opposed to paths from the root to specific objects.
Chapter 1 Introduction

When integrating information, full disjunctions are rather limited, since they are based solely on equalities among attributes, as in natural joins and equijoins. In many cases, there is a need to integrate information based on conditions that are more general than merely equalities among attributes. We show how our approach leads to a method of integrating information by means of general types of conditions and not just equality conditions. The work that is presented in Chapter 6 has been published in [44].

Related Work

A somewhat similar approach to ours is the approach of searching in XML documents. Information retrieval deals with the problem of searching for relevant documents in a repository of documents. When searching in XML repositories it is frequently not enough to find relevant documents, since documents are large and may contain a massive amount of data. The user should receive the most relevant fragments of documents rather then whole documents. This problem has been the focus of several papers. These papers made an attempt to combine the paradigm of querying with the paradigm of searching [21, 26, 38]. However, in a search, the result does not always include all the answers to the query. In addition, the result of a search may include items that are not an answer to the query. Thus, searching is different from querying.

A lot of research has been done, over the years, on querying incomplete data. The main issues that were considered are representation of missing information and query evaluation. In logical databases, one could use either a proof-theoretic approach [56] or a model-theoretic approach [62]. If there is some partial knowledge about the missing information, then it could be represented in the form of conditions that restrict the possible values of unknown data items [40]. The complexity of query evaluation in these approaches is investigated in [3, 40, 62].

The outerjoin [20, 29, 45] was introduced in order to preserve tuples that are lost during an ordinary join. Since outerjoins are not associative, it is rather difficult to formulate queries that use them. Moreover, the lack of associativity restricts the
ability to optimize queries with outerjoins. Efficient evaluation methods for outerjoin queries is the focus of several papers [12, 19, 33, 57].

Full disjunctions were proposed by Galiando-Legaria [34] as an alternative to outerjoins, since full disjunctions are commutative and associative. Full disjunctions bear some similarity to the weak instances that were investigated extensively in the framework of the universal-relation model [37, 39, 58, 59]. In both approaches, all the tuples of the base relations are preserved. In full disjunctions, tuples are joined together whenever possible. The joins should be connected in order to avoid combinations of tuples that are evidently not related. In weak instances, tuples are joined together as implied by the dependencies and queries can be evaluated by taking the union of all lossless joins [58, 59].

Different notions of acyclicity for hypergraphs were investigated in [31], including $\alpha$-acyclicity and $\gamma$-acyclicity. Properties of acyclic database schemes were investigated in [11]. (Note that a set of relation schemes can be represented as a hypergraph by viewing the attributes as nodes and viewing each relation scheme as a hyperedge.) Acyclicity leads to another form of query evaluation over universal relations [32].

Rajaraman and Ullman [55] enunciated the importance of full disjunctions in the context of integrating information from the Web. They showed how full disjunction could be used to find maximal join-consistent connections in trees of the OEM model of [53, 54]. Their main result is that a full disjunction can be evaluated by a natural outerjoin sequence if and only if the relation schemes are connected and $\gamma$-acyclic. Their algorithm was implemented in the Information Manifold System [47, 48] that integrates information from the Web.
Chapter 2

The Data Model

In this chapter, we present the data model and the query formulation of our work.

2.1 Databases

The data model that we use in this work is the semistructured data model, which is based on the Object Exchange Model (OEM) of [53]. The data are represented by rooted labeled directed graphs. Each node represents an object and has an identifier (oid). Values are attached to atomic nodes (i.e., nodes without outgoing edges). For simplicity, we assume that all atomic nodes are of type PCDATA (i.e., string). Nodes with outgoing edges represent complex objects. A database has a root and every node in the database is reachable from the root.

Definition 2.1 (Database) Let \( \mathcal{L} \) be a set of labels. A database \( D \) over \( \mathcal{L} \) is a 4-tuple \( D = (O, E_D, r_D, \alpha) \) that satisfies the following conditions. The set \( O \) is a finite set of objects. The set \( E_D \) is a finite set of 3-tuples \( (s, t, l) \), called labeled edges. In an edge, \( s \) and \( t \) are objects of \( O \), called the source and target of the edge, respectively; and \( l \) is an element of \( \mathcal{L} \), called the label on the edge. The object \( r_D \) is the root, and every object of \( O \) is reachable from the root via a directed path in \( D \). Finally, \( \alpha \) is a function that maps each atomic node to a value. The set of labels \( \mathcal{L} \) on the edges of \( E_D \) is called the ontology of \( D \).

Example 2.2 Figure 2.1 shows a movie database. Nodes are depicted as circles and the oid of each node appears inside the circle. The values that are attached to
Figure 2.1: A movie database with information on movies, T.V. Series, actors and directors.

atomic nodes are shown below those nodes, in boldface. Labels appear next to the edges. The ontology of the movie database is the set: \{Movie, T.V. Series, Actor, Director, Name, Title, Year\}.

2.2 Queries

Like databases, queries are also represented as labeled directed graphs and are similar to the queries defined in [41]. Nodes of queries represent variables rather than objects, and no value is associated with any node.

**Definition 2.3 (Query)** A query is 3-tuple \(Q = (V, E_Q, r_Q)\), where \(V\) is a finite set of variables, \(E_Q\) is a set of labeled edges and \(r_Q\) is the root of the query.

Queries are posed to databases. The result of posing a query to a database consists of assignments of database objects to query variables. Queries impose constraints according to their semantics. Thus, a query result comprises of assignments that satisfy the imposed constraints.
Figure 2.2: Three different queries for finding, in the movie database, pairs of an actor and a movie, such that the actor acted in the movie. For actors, the queries return their names. For movies, the queries return the title and the year of production.

**Definition 2.4 (Matching)** Suppose that \( Q = (V, E_Q, r_Q) \) is a query and \( D = (O, E_D, r_D, \alpha) \) is a database. A satisfying assignment (or a matching) of \( Q \) with respect to \((w, r.t.)\) \( D \) is a mapping \( \mu: V \rightarrow O \) that satisfies the constraints imposed by \( Q \).

In the following chapters, we will present several types of query semantics. In each semantics, queries impose different constraints. Now, we introduce two types of constraints for a query \( Q \). In the next two definitions, \( D \) denotes a database and \( \mu \) denotes an assignment of \( Q \) w.r.t. \( D \).

**Definition 2.5 (Root Constraint)** A root constraint (rc) is satisfied by matchings that map the root of \( Q \) to the root of \( D \).

**Definition 2.6 (Edge Constraint)** An edge constraint (ec) is written as \( ulv \), where \( ulv \) is an edge of \( Q \). The ec \( ulv \) is satisfied by the mapping \( \mu \) if \( D \) has an edge labeled with \( l \) from \( \mu(u) \) to \( \mu(v) \).

The usual semantics of queries is defined in terms of rigid matchings. In a rigid matching, two query variables that are connected by an edge with a label \( l \) are mapped to database objects that are also connected by an edge with the label \( l \).
Table 2.1: The rigid matchings of Query 1 and Query 2 (Figure 2.2) w.r.t. the movie database of Figure 2.1. For Query 3, there are no rigid matchings w.r.t. the movie database. The columns of the table are the query variables and the rows are matchings. Under each column there are the oid’s that are assigned to the variable that the column represents. For atomic objects, their values are presented, in addition to the oid.

**Definition 2.7 (Rigid Matching)** Consider a query Q and a database D. A rigid matching of Q w.r.t. D is an assignment that satisfies the rc and all the ec’s of Q. The set of all rigid matchings of Q w.r.t. D is denoted as \( M_r(Q, D) \).

Note that if \( \mu \) is a rigid matching of a query Q w.r.t. a database D, then there is a subgraph \( G \) of D such that the nodes of \( G \) are the objects assigned by \( \mu \) to the variables of Q and \( G \) is isomorphic to Q.

**Example 2.8** In Figure 2.2, three different queries are depicted. All the tree queries aimed at finding pairs of an actor and a movie such that the actor acted in the movie. For actors, the queries return their names. For movies, the queries return the title and the year of production. Without any knowledge about the structure of the database, it is impossible to tell whether among the three queries there is one that returns all the required actor-movie pairs. It is also impossible to say if one of these queries is better than the others, i.e., finds more actor-movie pairs than the other two queries. Table 2.1 presents the matchings that are produced when the queries of Figure 2.2 are posed to the movie database. Note that none of the queries return all the required pairs.
Chapter 3

Flexible and Semiflexible Queries

In this chapter, we present two new query semantics. The new semantics facilitate querying of databases where the hierarchy between objects is unknown or irregular. The outline of this chapter is as follows. Section 3.1 defines the flexible and the semiflexible semantics. Section 3.2 provides complexity results for query evaluation under the semiflexible semantics. Section 3.3 gives complexity results for query evaluation under the flexible semantics. The running times of query-evaluation algorithms are cast as functions of the combined size of the query, the database and the result. This approach of analyzing the complexity w.r.t. the size of the query, the database and the result (rather than the more common data complexity) can discern between cases that are computationally hard, even when the result is small, and cases where the running time is governed merely by the size of the result. Section 3.4 gives characterizations and complexity results for containment and equivalence of queries. Under the semiflexible and flexible semantics, two databases could be equivalent (even in nontrivial cases) in the sense that they give the same result for all queries. Section 3.5 characterizes equivalence of two databases. It also characterizes when a given database is equivalent to a tree and gives an algorithm for transforming the database to a tree. This result is important in the context of query evaluation, since queries over tree databases can be evaluated more efficiently, as shown in Section 3.2. Finally, in Section 3.6, we summarize the contributions of this chapter.
3.1 Flexible and Semiflexible Matchings

Before we define the new semantics, we need to introduce two new types of constraints. In the next two definitions, let $Q$ be a query and $D$ be a database.

**Definition 3.1 (Weak Edge Constraint)** A weak edge constraint (wec) is written as $lv$, where variable $v$ of $Q$ has an incoming edge labeled with $l$. The wec $lv$ is satisfied by a mapping $\mu$ if $\mu(v)$ has an incoming edge labeled with $l$.

**Definition 3.2 (Quasi Edge Constraint)** A quasi edge constraint (qec) is written as $ulv$, where $u$ and $v$ are variables of $Q$ and $l$ is the label of an edge from $u$ to $v$. The qec $ulv$ is satisfied by a mapping $\mu$ if $D$ either has a path from $\mu(u)$ to $\mu(v)$ or vice-versa.

Other types of constraints also include filtering constraints which are essentially selections. Considering filtering constraints is beyond the scope of this work. Some of the techniques developed in [41] could be used to extend our results also to queries with filtering constraints.

We introduce some notations. An edge from node $u$ to node $v$ that is labeled with $l$ is denoted as $ulv$. A path from node $u$ to node $v$ that ends with a label $l$ is denoted as $u*lv$. This notation is generalized in the natural way. For example, $ulv*kw$ denotes a path that starts at $u$, continues to $v$ through an edge labeled with $l$, and then continues to $w$ through a path that ends with the label $k$.

Next, we introduce two new notions of matchings. The assignment $\mu$ is a **semiflexible matching** if it satisfies the rc of $Q$ as well as the following two conditions.

1. For each finite path $v_0l_1v_1l_2v_2\cdots l_nv_n$ of $Q$, where $v_0$ is the root, the images $\mu(v_i)$ $(0 \leq i \leq n)$ lie on some path $\phi$ of $D$, such that for all $i$ $(1 \leq i \leq n)$, the edge of $\phi$ that enters $\mu(v_i)$ is labeled with $l_i$. Note that $\mu(v_0)$ is the root of $D$ and, hence, is the first node on $\phi$, but the rest of the $\mu(v_i)$ are not necessarily arranged on $\phi$ in the order $\mu(v_1), \ldots, \mu(v_n)$. Moreover, other nodes could be on $\phi$ between the $\mu(v_i)$.

2. For every strongly connected component $C$ in $Q$, the image of $C$ is a strongly connected component in $D$. 
A strongly connected component in a graph is a set of nodes, such that between every two nodes there is a path in both directions. The preservation of the strongly connected components is needed in order to deal with paths that have cycles.

Condition 1 above can be defined equivalently as follows.

**Definition 3.3 (SF-Condition)** A path \( v_0l_1v_1l_2v_2 \cdots l_nv_n \) of \( Q \) satisfies the SF-Condition w.r.t. \( \mu \) if there is a permutation \( \sigma \) of \( 0, \ldots, n \), such that \( \sigma(0) = 0 \) and \( D \) has a path of the following form:

\[
\mu(v_{\sigma(0)})l_{\sigma(1)}\mu(v_{\sigma(1)})l_{\sigma(2)}\mu(v_{\sigma(2)}) \cdots l_{\sigma(n)}\mu(v_{\sigma(n)}).
\]

Now we can give an equivalent definition of a semiflexible matching.

**Definition 3.4 (Semiflexible Matching)** An assignment \( \mu \) of a query \( Q \) w.r.t. a database \( D \) is a semiflexible matching if the next three conditions hold.

1. The assignment \( \mu \) satisfies the rc of \( Q \).
2. Every finite path \( v_0l_1v_1l_2v_2 \cdots l_nv_n \) of \( Q \), where \( v_0 \) is the root, satisfies the SF-Condition w.r.t. \( \mu \).
3. For every set \( C = \{v_{i_1}, \ldots, v_{i_m}\} \), if \( C \) is a strongly connected component in \( Q \), then the set \( \{\mu(v_{i_1}), \ldots, \mu(v_{i_m})\} \) is a strongly connected component in \( D \).

We denote the set of all semiflexible matchings of \( Q \) w.r.t. \( D \) as \( \mathcal{M}_s(Q,D) \).

The second new notion of matchings that we introduce in this chapter is the notion of flexible matching that is defined next.

**Definition 3.5 (Flexible Matching)** A flexible matching of a query \( Q \) w.r.t. a database \( D \) is an assignment \( \mu \) for which the following three conditions hold.

1. The rc is satisfied by \( \mu \).
2. All the wc's of \( Q \) are satisfied by \( \mu \).
3. All the gec's of \( Q \) are satisfied by \( \mu \).

We denote the set of all flexible matchings of \( Q \) w.r.t. \( D \) as \( \mathcal{M}_f(Q,D) \).
Chapter 3  Flexible and Semiflexible Queries

Intuitively, in both the semiflexible and flexible semantics, we assume that nodes of $D$ that are connected by a path are semantically related. The difference is that in the flexible semantics, we also assume that this relationship is transitive. This difference entails another important difference between the semiflexible and the flexible semantics. In the flexible semantics, cycles in the query are not necessarily mapped to cycles in the database. Semiflexible matchings, however, require strongly connected components in the query to be mapped to strongly connected components in the database.

**Proposition 3.6 (Increasing Flexibility)** Given a database $D$ and a query $Q$, the following holds: $\mathcal{M}_r(Q,D) \subseteq \mathcal{M}_{sf}(Q,D) \subseteq \mathcal{M}_f(Q,D)$.

**Proof.** Let $\mu$ be a rigid matching of $Q$ w.r.t. $D$. It is easy to see that $\mu$ is also a semiflexible matching of $Q$ w.r.t $D$. First, $\mu$ satisfies the rc of $Q$. Second, consider a finite path $\phi$ in $Q$ that starts at the root. By taking the identity permutation, it is easy to see that $\phi$ satisfies the SF-Condition w.r.t. $\mu$ because $\mu$ satisfies all the ec's of $Q$. Third, if $C = \{v_1, \ldots, v_m\}$ is a strongly connected component in $Q$, then $\{\mu(v_1), \ldots, \mu(v_m)\}$ is a strongly connected component in $D$, because $\mu$ satisfies all the ec's of $Q$.

Let $\mu$ be a semiflexible matching of $Q$ w.r.t. $D$. We show that $\mu$ is also a flexible matching of $Q$ w.r.t. $D$. First, $\mu$ satisfies the rc of $Q$. Secondly, we show that $\mu$ satisfies all the wec's in $Q$. Given a wec $lv$, since $v$ is reachable from the root of $Q$, there is a path $v_0l_1v_1 \cdots l_nv_n$ in $Q$ such that $v_0$ is the root of $Q$, $l_n$ is $l$ and $v_n$ is $v$. This path satisfies the SF-Condition w.r.t. $\mu$. Thus, there is a permutation $\sigma$ such that $D$ contains the path $\mu(v_{\sigma(0)})^{l_{\sigma(1)}}\mu(v_{\sigma(1)})^{l_{\sigma(2)}}\mu(v_{\sigma(2)})^{l_{\sigma(3)}}\cdots l_{\sigma(n)}\mu(v_{\sigma(n)})$. In this path, $l_{\sigma(i)}\mu(v_{\sigma(i)})$, where $1 \leq i \leq n$ and $\sigma(i) = n$, shows that $\mu$ satisfies the wec $lv$.

Thirdly, we show that $\mu$ satisfies all the qec's in $Q$. Given a qec $ulv$ in $Q$, since $u$ is reachable from the root of $Q$, there is a path $v_0l_1v_1 \cdots l_{n-1}v_{n-1}l_nv_n$ in $Q$ such that $v_0$ is the root of $Q$, $v_{n-1}$ is $u$, $l_n$ is $l$ and $v_n$ is $v$. This path satisfies the SF-Condition, so there is a path in $D$ that contains both $\mu(v)$ and $\mu(u)$. Thus, either there is a path in $D$ from $\mu(v)$ to $\mu(u)$ or vice-versa.  \[\square\]
Some languages for semistructured data use regular expressions as a tool for formulating queries when the structure is not completely known to the user. In principle, the expressive power of regular expressions subsumes the semiflexible and flexible semantics. However, in addition to regular expressions, one may also need unions and joins in order to fully express these semantics. In any case, the semiflexible and flexible semantics facilitate a much more succinct and easier style of writing queries. Moreover, we will show that evaluation and optimization of semiflexible and flexible queries are easier than evaluation and optimization of queries with regular expressions.

3.1.1 Examples

A movie database that holds information on movies, T.V. series, actors and directors is depicted in Figure 2.1. Suppose that Alice wants to query the data. She is familiar with the ontology of the database, but does not know how the information is organized. We examine a few cases.

Example 3.7 Alice tries to look in the database for information on movies. She
Table 3.1: The flexible matchings for Query 3 of Figure 3.1(c) and the database of Figure 2.1.

assumes that the information on each actor includes the movies in which the actor participated. Thus, she formulates Query 1 of Figure 3.1(a). There are no rigid matchings of Query 1 w.r.t. the movie database. However, under the semiflexible semantics, the answer of Query 1 will include a mapping of $r$, $u$ and $x$ to 1, 14 and 29, respectively, even though 14 and 29 are not adjacent. The answer will also include a mapping of $r$, $u$ and $x$ to the next object triplets: (1) to 1, 21 and 11; (2) to 1, 22 and 11; and (3) to 1, 25 and 12. Note that in the last three mappings the movie is above the actor while in the query the actor is above the movie.

Example 3.8 Alice wants to find two actors who played in the same movie. She formulates Query 2 of Figure 3.1(b). Obviously, there are no rigid matchings in this case since the database is a tree and the query is a dag. However, under either the semiflexible or the flexible semantics, Alice will find Mark Hamill and Harrison Ford as two actors who played in the same movie (Star Wars). This is due to the fact that assigning $r, u, v, x$ to 1, 21, 22, 11, respectively, is a semiflexible (and, hence, also a flexible) matching of the query w.r.t. the movie database.

Example 3.9 When Query 3 of Figure 3.1(c) is posed to the movie database as a rigid query, there are no matchings of the query w.r.t. the database. Under the semiflexible semantics, the query has only a single matching ($r, u, v, w, x, y$ and $z$ are mapped to 1, 14, 27, 29, 35, 42 and 52, respectively), although the database has two movies with a director. However, under the flexible semantics, we get the assignments that are shown in Table 3.1. For atomic nodes, the value is presented in addition to the oid.
3.1.2 More on Semiflexible Matchings

Consider a query $Q$, a database $D$ and an assignment $\mu$ of $Q$ w.r.t. $D$. Obviously, checking whether $\mu$ is a flexible matching can be done in polynomial time in the size of the query and the database. This follows from the next three facts. First, the number of qec's and the number of wec's, in a query, is equal to the number of query edges. Second, checking satisfaction of the root constraint and satisfaction of the wec's is immediate, i.e., in $O(1)$. Third, checking satisfaction of a qec requires to check that two database nodes are connected by a path and this can be done in linear time in the size of the database.

It is not obvious how to check in polynomial time whether an assignment to the variables of a query $Q$ is a semiflexible matching. The reason is that all the paths of $Q$ have to be considered. If $Q$ is a dag (directed acyclic graph), the number of paths could be exponential, and if $Q$ has a cycle, then there are infinitely many paths. In this section, we show that for any given query $Q$, checking whether an assignment is a semiflexible matching could be done in polynomial time in the size of $Q$ and $D$. For the case of a cyclic query, we first show that it is sufficient to consider only simple paths and simple cycles in order to check whether a given assignment is a semiflexible matching. Based on this result, we provide sufficient and necessary conditions that can be checked in polynomial time in the size of the query and the database, such that if the conditions hold then the assignment is a semiflexible matching. Note that as a matter of terminology, a cyclic query (database) has at least one cycle. Thus, the class of cyclic queries (databases) does not include any dag query (database). A tree query (database), however, is a special case of a dag query (database).

Consider a query $Q$, a database $D$ and an assignment $\mu$ of $Q$ w.r.t. $D$. The next definition provides a characterization for satisfiability of the SF-Condition w.r.t. $\mu$ by simple paths of $Q$. This characterization holds even when the database is cyclic.

**Definition 3.10 (Connectivity-Preserving Mapping)** Let $Q$ be a query and $D$ be a database. We say that the assignment $\mu : V \rightarrow O$ is a connectivity-preserving mapping if it satisfies the following conditions.
• For all labels $l$, all nodes $v$ and the root $r$ of $Q$, if $Q$ has a simple path of the form $r*l_0$, then $D$ has a path of the form $\mu(r)*l_\mu(v)$; and

• For all labels $l$ and $k$, and all nodes $u$, $v$ and $w$ of $Q$, if $Q$ has a simple path of the form $ulv*kw$, then $D$ either has a path of the form $\mu(v)*k\mu(w)$ or a path of the form $\mu(w)*l_\mu(v)$.

The next lemma shows that the conditions in Definition 3.10 are necessary and sufficient for satisfiability of the SF-Condition w.r.t. $\mu$ by simple paths of $Q$.

**Lemma 3.11 (The SF-Condition for Simple Paths)** Consider a query $Q$, a database $D$ (that may have cycles) and an assignment $\mu: V \rightarrow O$ of $Q$ w.r.t. $D$. Every simple path in $Q$ satisfies the SF-Condition w.r.t. $\mu$ if and only if $\mu$ is a connectivity-preserving mapping.

**Proof.** It is easy to see that if every simple path in $Q$ satisfies the SF-Condition w.r.t. $\mu$, then $\mu$ is a connectivity-preserving mapping.

Before proving the other direction, we present and prove a claim regarding the existence and non-existence of specific paths in a strongly connected component.

**Claim 3.12** Consider a wec’s $l_1v_1, \ldots, l_nv_n$, a database $D$ and a mapping $\mu$ of $v_1, \ldots, v_n$ to objects of $D$ such that $\mu(v_1), \ldots, \mu(v_n)$ are all in a strongly connected component of $D$. Suppose that for each two wec’s $l_i v_i$ and $l_j v_j$, $D$ either has a path of the form $\mu(v_i)*l_i \mu(v_j)$ or a path of the form $\mu(v_j)*l_i \mu(v_i)$. If there are $v_a$ and $v_b$, among $v_1, \ldots, v_n$, such that $D$ has no path of the form $\mu(v_b)*l_a v_a$ then the following three assertions will hold.

1. For each $1 \leq i \leq n$, $D$ has no path of the form $\mu(v_i)*l_a \mu(v_a)$.

2. For each $1 \leq i \leq n$, $D$ has a path of the form $\mu(v_a)*l_i \mu(v_i)$.

3. For each $1 \leq i \leq n$ and $1 \leq j \leq n$, if $v_i \neq v_a$ then $D$ has a path of the form $\mu(v_j)*l_i \mu(v_i)$.

We show correctness of the first assertion. Suppose that there was $1 \leq i \leq n$ such that $D$ has a path of the form $\mu(v_i)*l_a \mu(v_a)$. This path can be combined with a path
from $\mu(v_b)$ to $\mu(v_i)$ (such a path must exist in a strongly connected component) and this yields a contradiction to the assumption that $D$ has no path of the form $\mu(v_b) * l_a v_a$.

The correctness of the second assertion follows from the first assertion and the assumption that for each two wec’s $l_i v_i$ and $l_j v_j$, $D$ either has a path of the form $\mu(v_i) * l_j \mu v_j$ or a path of the form $\mu(v_j) * l_i \mu(v_i)$.

We show the correctness of the third assertion. For each $1 \leq j \leq n$, $D$ has a path from $\mu(v_j)$ to $\mu(v_a)$, since $\mu(v_j)$ and $\mu(v_a)$ are two objects on a strongly connected component. According to the second assertion, $D$ has a path of the form $\mu(v_a) * l_i \mu(v_i)$. Combining these two paths yields a path of the form $\mu(v_j) * l_i \mu(v_i)$.

In the notations of Claim 3.12, we call $\mu(v_a)$ the entry point of $\mu(v_1), \ldots, \mu(v_n)$. It is easy to see that under the conditions of the claim, $\mu(v_1), \ldots, \mu(v_n)$ cannot have more than one entry point. Specifically, if one of the objects, $\mu(v_r)$, is the database root, then for each $v_j$ other than the root, $D$ has a path of the form $\mu(v_r) * l_j \mu(v_j)$. In this case, $\mu(v_r)$, i.e., the database root, is the entry point of $\mu(v_1), \ldots, \mu(v_n)$.

Now, we can prove the other direction of the lemma. Suppose that $\mu$ is a connectivity-preserving mapping and consider a simple path $\pi = v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of $Q$. We need to show that for some permutation $\sigma$, the database $D$ has a path of the form

$$\mu(v_{\sigma(0)}) * l_{\sigma(1)} \mu(v_{\sigma(1)}) * l_{\sigma(2)} \mu(v_{\sigma(2)}) \cdots * l_{\sigma(n)} \mu(v_{\sigma(n)})$$

As a preliminary step, we partition the nodes $\mu(v_0), \mu(v_1), \ldots, \mu(v_n)$ according to the maximal strongly connected components of $D$. Let $C_1, \ldots, C_m$ be the maximal strongly connected components sorted topologically (see [27] for topological-sort algorithms).

We choose a permutation $\sigma$ of $0, \ldots, n$ that has the following properties. First, $\sigma(0) = 0$. Second, the sequence of objects $\mu(v_{\sigma(0)}), \mu(v_{\sigma(1)}), \ldots, \mu(v_{\sigma(n)})$ conforms to the topological ordering of $C_1, \ldots, C_m$. That is, if $\mu(v_{\sigma(i)}) \in C_t$, $\mu(v_{\sigma(j)}) \in C_t$ and $i < j$ then $t_1 \leq t_2$. Third, if $\mu(v_{\sigma(i_1)}), \ldots, \mu(v_{\sigma(i_k)})$ are objects in a strongly connected component of $D$ and $\mu(v_{\sigma(i_j)})$ is an entry point of $\mu(v_{\sigma(i_1)}), \ldots, \mu(v_{\sigma(i_k)})$ then $i_j \leq i_h$, for each $1 \leq h \leq k$. That is, among the objects $\mu(v_{\sigma(i_1)}), \ldots, \mu(v_{\sigma(i_k)})$,
\( \mu(v_{\sigma(i)}) \) appears first in the path that is defined by \( \mu \) and \( \sigma \). The order of the other objects is chosen arbitrarily. If there is no entry point of \( \mu(v_{\sigma(i_1)}), \ldots, \mu(v_{\sigma(i_k)}) \) then the order of all these objects is chosen arbitrarily, i.e., there can be any order for \( i_1, \ldots, i_k \). Note that we can choose such \( \sigma \) because \( C_1, \ldots, C_m \) are maximal strongly connected components and they are sorted topologically.

Note that \( v_{\sigma(0)} \) and \( \mu(v_{\sigma(0)}) \) are the roots of \( Q \) and \( D \), respectively. Moreover, if both \( \mu(v_{\sigma(i-1)}) \) and \( \mu(v_{\sigma(i)}) \) \( (1 \leq i \leq n) \) lie on the same path \( \phi \) of \( D \), then one of the next two conditions must be true.

- The path \( \phi \) is from \( \mu(v_{\sigma(i-1)}) \) to \( \mu(v_{\sigma(i)}) \).
- The objects \( \mu(v_{\sigma(i-1)}) \) and \( \mu(v_{\sigma(i)}) \) both belong to the same strongly connected component of \( D \).

We claim that \( D \) has a path of the form

\[
\mu(v_{\sigma(0)}) * l_{\sigma(1)} * \mu(v_{\sigma(1)}) * l_{\sigma(2)} * \mu(v_{\sigma(2)}) * \ldots * l_{\sigma(n)} * \mu(v_{\sigma(n)}).
\]

To prove the claim, we show that for all \( 1 \leq i \leq n \), \( D \) has a path of the form \( \mu(v_{\sigma(i-1)}) * l_{\sigma(i)} * \mu(v_{\sigma(i)}) \). For \( i = 1 \), node \( v_{\sigma(0)} \) is the root of \( Q \), and so, \( Q \) has a path of the form \( v_{\sigma(0)} * l_{\sigma(1)} * v_{\sigma(1)} \). Thus, by the first part of the condition in Definition 3.10, \( D \) has a path of the form \( \mu(v_{\sigma(0)}) * l_{\sigma(1)} * \mu(v_{\sigma(1)}) \).

For \( i > 1 \), there are three cases to be considered. First, we consider the case where \( \mu(v_{\sigma(i-1)}) \) and \( \mu(v_{\sigma(i)}) \) belong to the same strongly connected component. According to the construction of \( \sigma \), \( \mu(v_{\sigma(i)}) \) is not the entry point of the nodes in the strongly connected component. This is because the entry point appears in the path before all the other nodes of the strongly connected component. Hence, according to the third assertion of Claim 3.12, \( D \) has a path of the form \( \mu(v_{\sigma(i-1)}) * l_{\sigma(i)} * \mu(v_{\sigma(i)}) \).

In the next two cases, \( \mu(v_{\sigma(i-1)}) \) and \( \mu(v_{\sigma(i)}) \) do not belong to the same strongly connected component. In one case, \( v_{\sigma(i-1)} \) appears before \( v_{\sigma(i)} \) on the path \( \pi \) of \( Q \). In the other case, \( v_{\sigma(i-1)} \) appears after \( v_{\sigma(i)} \) on \( \pi \).

If \( v_{\sigma(i-1)} \) appears first, then \( Q \) has a path of the form \( v_{\sigma(i-1)} * l_{\sigma(i)} * v_{\sigma(i)} \). Thus, according to the second part of the condition in Definition 3.10, \( D \) must have a path of the form \( \mu(v_{\sigma(i-1)}) * l_{\sigma(i)} * \mu(v_{\sigma(i)}) \). Alternatively, if \( v_{\sigma(i)} \) appears first, then there is
a $j$ ($0 \leq j \leq n - 2$), such that $Q$ has a path of the form $v_j l_{\sigma(i)} v_{\sigma(i)}^* l_{\sigma(i-1)} v_{\sigma(i-1)}$. Thus, according to the second part of the condition, $D$ must have a path of the form $\mu(v_{\sigma(i-1)}) l_{\sigma(i)} (v_{\sigma(i)})$. \hfill \Box

Consider a query $Q$, a database $D$ and an assignment of $Q$ w.r.t. $D$.Lemma 3.11 provides a test for checking if a simple path $\pi$ in $Q$ satisfies the SF-Condition. In this test, we just need to check that the conditions of Definition 3.10 are satisfied w.r.t. $\pi$. It is easy to see that the test can be done in polynomial time in the size of $\pi$ and $D$.

**Corollary 3.13 (DAG Queries)** Consider a dag query $Q$ and a database $D$ that may have cycles. The assignment $\mu : V \rightarrow O$ is a semiflexible matching of $Q$ w.r.t. $D$ if and only if (1) $\mu$ is a connectivity-preserving mapping, and (2) $\mu$ maps the query root to the database root.

**Proof.** If $\mu$ is a semiflexible matching of $Q$ w.r.t. $D$ then, by Lemma 3.11, $\mu$ is a connectivity-preserving mapping; and from the first condition of Definition 3.4 follows that $\mu$ maps the query root to the database root.

For the other direction, since $Q$ is a dag, we only need to show that $\mu$ satisfies the first two conditions of Definition 3.4. Condition 1 of Definition 3.4 requires satisfaction of the rc. The condition holds because $\mu$ maps the query root to the database root. Condition 2 requires satisfaction the SF-condition w.r.t. $\mu$, for every finite path in $Q$ and Lemma 3.11 proves that this condition holds. \hfill \Box

From Corollary 3.13 it follows that when $Q$ is a dag, the complexity of checking whether $\mu$ is a semiflexible matching is in polynomial time in the size of $Q$ and $D$.

Next, we will consider cyclic queries. Since cyclic queries have infinitely many paths, a naive test based on the definition of a semiflexible matching does not terminate. The following lemma alleviates this difficulty.

The path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ (where $v_0$ is not necessarily the root) is a simple path if all the $v_i$ are distinct. It is a cycle if $v_0 = v_n$, and it is a simple cycle if $v_0 = v_n$ and $v_1, v_2, \ldots, v_n$ are distinct.
Lemma 3.14 (Condition for Cyclic Queries) Let Q be a cyclic query and D be a cyclic database. The assignment \( \mu \) is a semiflexible matching of Q w.r.t. D if and only if the following conditions are satisfied.

1. The assignment \( \mu \) satisfies the rc of Q.

2. All the simple paths of Q that start at the root satisfy the SF-Condition w.r.t. \( \mu \).

3. For all the simple cycles \( u_0l_1u_1 \cdots l_ku_k \) of Q (where \( u_0 \) is not necessarily the root), D has a cycle (which is not necessarily a simple cycle) of the form \( \mu(u_0) \ast l_1 \mu(u_1) \cdots \ast l_k \mu(u_k) \).

Proof. If \( \mu \) is a semiflexible matching, then, by Definition 3.4, Condition 1 and Condition 2 hold. Next, we will show the necessity of Condition 3.

Let \( \mu \) be a semiflexible matching of Q w.r.t. D, and let \( C = u_0l_1u_1 \cdots l_ku_k \), where \( u_0 = u_k \), be a simple cycle of Q.

According to the definition of a semiflexible matching (Definition 3.4), the set \( \{ \mu(u_0), \ldots, \mu(u_k) \} \) is contained in a strongly connected component of D, because \( \{ u_0, \ldots, u_k \} \) is contained in a strongly connected component of Q. Therefore, there is a path from \( \mu(u_i) \) to \( \mu(u_{i+1}) \) (\( 0 \leq i \leq k - 1 \)).

The second observation is that D has a path of the form \( \mu(u_i) \ast l_i \mu(u_i) \) (\( 1 \leq i \leq k \)). In proof, for each node \( u_i \) (\( 1 \leq i \leq k \)) in C, there is a path that starts at the root of Q, continues to a node in C and then goes to \( u_i \) through the edge \( u_{i-1}l_iu_i \). This path can be extended by going around the cycle C from \( u_i \) back to itself. Thus, Q has a path of the form \( r_Q \ast l_iu_i \ast l_iu_i \), where \( r_Q \) is the root of Q. This path satisfies the SF-Condition (Definition 3.3), since \( \mu \) is a semiflexible matching. Therefore, D must have a path of the form \( \mu(u_i) \ast l_i \mu(u_i) \) (\( 1 \leq i \leq k \)).

By combining the path from \( \mu(u_{i-1}) \) to \( \mu(u_{i}) \) with the path \( \mu(u_{i}) \ast l_i \mu(u_i) \) (\( 1 \leq i \leq k \)), it follows that D has a path of the form \( \mu(u_{i-1}) \ast l_i \mu(u_i) \) (\( 1 \leq i \leq k \)). Therefore, D also has a path of the form \( C_D = \mu(u_0) \ast l_1 \mu(u_1) \cdots \ast l_k \mu(u_k) \). Since \( u_0 = u_k \), it follows that \( \mu(u_0) = \mu(u_k) \) and, thus, \( C_D \) is a cycle in D. Thus, we have shown that Condition 3 is satisfied.

For the other direction, suppose that \( \mu \) is a mapping of the variables of Q to
Chapter 3  Flexible and Semiflexible Queries

objects of \( D \), such that conditions 1–3 are satisfied. By Condition 3, \( \mu \) maps strongly
connected components of \( Q \) to strongly connected components of \( D \). Next, we will
show that finite paths of \( Q \) satisfy the SF-Condition w.r.t. \( \mu \).

Let \( \pi = v_0l_1v_1 \cdots l_nv_n \) be a path of \( Q \), where \( v_0 \) is the root of \( Q \). We will show
by induction on \( n \) that \( \pi \) satisfies the SF-Condition w.r.t. \( \mu \).

For \( n = 1 \), \( \pi \) is either a simple path (if \( v_0 \neq v_1 \)) or a simple cycle (if \( v_0 = v_1 \)).
In the first case, \( \pi \) satisfies the SF-Condition by Condition 2, and in the second
case—by Condition 3.

Next, suppose that the claim holds for paths of length less that \( n \), where \( n > 1 \).
We have to show that the claim holds for the path \( \pi \).

Once again, if \( \pi \) is a simple path or a simple cycle, then either Condition 2 or
Condition 3 implies that \( \pi \) satisfies the SF-Condition.

If \( \pi \) is neither a simple path nor a simple cycle, then it must have variables \( v_i \) and
\( v_k \) (\( i < k \)), such that the path \( \pi_c = v_il_{i+1}v_{i+1} \cdots l_kv_k \) is a simple cycle, i.e., \( v_i = v_k \).
By Condition 3, \( D \) has a cycle of the form \( \phi_c = \mu(v_i)l_{i+1}\mu(v_{i+1}) \cdots l_k\mu(v_k) \).

By the induction hypothesis, the path
\[
\pi_p = v_0l_1v_1 \cdots l_iv_i l_{i+1}v_{k+1} \cdots l_nv_n
\]
satisfies the SF-Condition. Thus, there is a permutation \( \sigma_p \) of \( 0, 1, \ldots, i, k+1, \ldots, n \),
such that \( \sigma_p(0) = 0 \) and \( D \) has a path of the following form.

\[
\phi_p = \mu(v_{\sigma_p(0)})l_{\sigma_p(i)}\mu(v_{\sigma_p(i)}) \cdots
\]

\[
* l_{\sigma_p(i)}\mu(v_{\sigma_p(i)})l_{\sigma_p(k+1)}\mu(v_{\sigma_p(k+1)}) \cdots
\]

\[
* l_{\sigma_p(n)}\mu(v_{\sigma_p(n)})
\]

We will now show that the cycle \( \phi_c \) and the path \( \phi_p \) can be combined into a
single path \( \phi \) that shows that \( \pi \) satisfies the SF-Condition. Intuitively, this is done
by inserting the cycle \( \phi_c \) into the path \( \phi_p \) starting at an occurrence of \( v_i \) in \( \phi_p \). To
do that, we need to define a permutation \( \sigma \) of \( 0, 1, \ldots, n \) that combines the effect of
the permutation \( \sigma_p \) with the effect of the identity permutation of \( i+1, \ldots, k \), which
relates the simple cycle \( \pi_c \) to the cycle \( \phi_c \).
The permutation $\sigma_p$ is defined over $0, 1, \ldots, i, k + 1, \ldots, n$, i.e., it has a gap between $i + 1$ and $k$. However, this gap is not necessarily the place to put the cycle $\phi_c$. Therefore, we first have to shift some positions in order to create the gap in the right place.

Formally, there is a $j$, such that $\sigma_p(j) = i$. Note that $j$ is the position of $\mu(v_i)$ in the path $\phi_p$. There are two cases for creating the gap, starting at position $j + 1$, and for each case $\sigma$ is defined differently.

If $j + 1 \leq i$, then the gap has to be created by shifting right; that is, for $j + 1 \leq m \leq i$, we define $\sigma(m + k - i) = \sigma_p(m)$.

If $k + 1 \leq j$, then the gap has to be created by shifting left; that is, for $k + 1 \leq m \leq j$, we define $\sigma(m - (k - i)) = \sigma_p(m)$.

Now, we fill the gap with the cycle $\phi_c$ by defining $\sigma(j + m) = i + m$ for $1 \leq m \leq k - i$. Finally, we complete the definition of $\sigma$ by defining $\sigma(m) = \sigma_p(m)$, for all other values of $m$.

It thus follows that $D$ has a path $\phi$ of the form

$$\mu(v_{\sigma(0)}) * l_{\sigma(1)} \mu(v_{\sigma(1)}) * l_{\sigma(2)} \mu(v_{\sigma(2)}) \cdots * l_{\sigma(n)} \mu(v_{\sigma(n)})$$

and this path shows that $\pi$ satisfies the SF-Condition.

Lemma 3.14 shows that when $Q$ is a cyclic query, verifying that a matching is a semiflexible matching is decidable. However, the verification requires to check that the image of every simple cycle in $Q$ is on a cycle in $D$. Since the number of simple cycles in a query can be exponential in the size of the query, verification according to Lemma 3.14 may require exponential time. The next definition provides a condition that can be verified in polynomial time in the size of the query and the database. This condition will assist in checking if simple cycles of the query are mapped to cycles in the database.

**Definition 3.15 (Cycle-Preserving Mapping)** Let $Q$ be a cyclic query, $D$ be a cyclic database and $\mu : Q \to D$ be an assignment of $Q$ w.r.t. $D$. We say that $\mu$ is a cycle-preserving mapping if for every pair of variable $v$ and $u$, if $Q$ has a path of the form $u * l v$ and a path of the form $v * l' u$, then $D$ has a path of the form $\mu(u) * l \mu(v)$ and a path of the form $\mu(v) * l' \mu(u)$. 

Chapter 3  Flexible and Semiflexible Queries

The next lemma shows that the condition in Definition 3.15 is sufficient and necessary for mapping simple cycles of the query into cycles of the database. Note that the two variables \( u \) and \( v \) in Definition 3.15 could be equal.

**Lemma 3.16 (Cycle Preserving)** Let \( Q \) be a cyclic query, \( D \) be a cyclic database and \( \mu : Q \to D \) be an assignment of \( Q \) w.r.t. \( D \). The following conditions are equivalent.

1. For every simple cycle \( u_0l_1u_1\cdots l_ku_k \) of \( Q \) (where \( u_0 \) is not necessarily the root), \( D \) has a cycle (which is not necessarily a simple cycle) of the form \( \mu(u_0)*l_1\mu(u_1)\cdots l_k\mu(u_k) \).

2. The assignment \( \mu \) is a cycle-preserving mapping.

**Proof.** Assume that Condition 2 holds and let \( u_0l_1u_1\cdots l_ku_k \) be a simple cycle in \( Q \). Then, for each pair of nodes \( u_i \) and \( u_{i+1} \) \((0 \leq i \leq k-1)\), \( Q \) has a path \( u_i*l_{i+1}u_{i+1} \) and a path \( u_{i+1}*l_iu_i \) (note that \( u_0 = u_k \), thus \( l_0 = l_k \)). By Definition 3.15, in \( D \) there is a path of the form \( \mu(u_i)*l_{i+1}\mu(u_{i+1}) \). This shows that \( D \) has a cycle of the form \( \mu(u_0)*l_1\mu(u_1)\cdots l_k\mu(u_k) \).

For the other direction, we start by proving the following claim.

**Claim 3.17** Given that Condition 1 holds, for every pair of variables \( u \) and \( v \) that are on a cycle (not necessarily a simple cycle), the images of \( u \) and \( v \), i.e., \( \mu(u) \) and \( \mu(v) \), are on a cycle in \( D \).

The proof of the claim is by an induction on the number of nodes in the cycle that contains \( u \) and \( v \). First, we consider the case where \( u \) and \( v \) are equal and the cycle is a loop, i.e., contains only one edge. In this case, the cycle is simple and has the form \( u\mu(u) \). By Condition 1, \( D \) has a cycle \( \mu(u)*l\mu(u) \) and the claim holds.

Secondly, we consider the case where the cycle contains only two nodes. In this case, the cycle has an edge from \( u \) to \( v \) and vice-versa. Thus, there is a simple cycle that contains \( u \) and \( v \). According to Condition 1, the image of this cycle w.r.t. \( \mu \) is on a cycle in \( D \).
Next, we show that if the claim holds for each $u$ and $v$ that are on a cycle with less than $k$ nodes then the claim also holds for each $u$ and $v$ that are on a cycle with $k$ nodes. Let $u$ and $v$ be nodes on a cycle $C$ with $k$ nodes. If $C$ is a simple cycle then, according to Condition 1, the nodes in the set $\{\mu(w) \mid w \text{ on } C\}$ are on a cycle in $D$ and, in particular, $\mu(u)$ and $\mu(v)$ are on a cycle in $D$. If $C$ is not a simple cycle then there is a node $w$, such that $w$ appears twice in $C$. At least one of the following three cases must hold.

1. $C$ is a path of the form $u*w*w*v*u$;

2. $C$ is a path of the form $u*v*w*w*u$;

3. $C$ is a path of the form $u*w*v*w*u$;

In all three cases, $C$ is a composition of two smaller cycles. In the first case, $Q$ has a cycle of the form $u*w*v*u$ with less than $k$ nodes and thus, by the induction hypothesis, $\mu(u)$ and $\mu(v)$ are on a cycle in $D$. The second case is similar to the first case. In the third case, $C$ contains the following two sub-cycles: A cycle $C_1$ of the form $u*w*u$ and a cycle $C_2$ of the form $v*w*v$. The size of each one of the two cycles is smaller than $k$. Thus, according to Condition 1, $D$ has a cycle that contains $\mu(u)$ and $\mu(w)$ and $D$ also has a cycle that contains $\mu(v)$ and $\mu(w)$. Since these two cycles have a shared node, $\mu(w)$, they can be combined to form a cycle that contains $\mu(u)$ and $\mu(v)$. This complete the proof of the claim.

Now, assume that Condition 1 holds and let $u$ and $v$ be variables of $Q$, such that there are paths of the form $u*l*v$ and $v*l'u$, in $Q$. It must hold that $Q$ has (1) a simple cycle $v*l*v$; (2) a simple cycle $u*l'u$; and (3) a cycle $v*u*v$. The reason for having the first (second) cycle is that $v$ ($u$) is on a cycle that goes into $v$ ($u$) through an edge labeled with $l$ ($l'$). From the existence of the first two cycles and from Condition 1, it follows that $D$ has a path of the form $\mu(v)*l*\mu(v)$ and a path of the form $\mu(u)*l'*\mu(u)$. From the third cycle and the claim above, it follows that $D$ has a cycle $\mu(v)*\mu(u)*\mu(v)$. Combining these cycles yields a cycle $\mu(v)*l'*\mu(u)*l*\mu(v)$. Therefore, $D$ has a path of the form $\mu(u)*l*\mu(v)$ and a path of the form $\mu(v)*l'*\mu(u)$. $\square$
Chapter 3 Flexible and Semiflexible Queries

Using the above lemmas, we can introduce a condition that is both necessary and sufficient for an assignment $\mu$ of a cyclic query w.r.t. a cyclic database to be a semiflexible matching. The condition requires local tests in the sense that only pairs of nodes are checked each time and the check for each pair can be done in polynomial time in the size of the query and the database.

**Corollary 3.18 (Cyclic Queries)** Suppose that $Q$ is a cyclic query and $D$ is a cyclic database. The assignment $\mu$ is a semiflexible matching of $Q$ w.r.t. $D$ if and only if the following conditions are satisfied.

1. The assignment $\mu$ satisfies the rc of $Q$.
2. The assignment $\mu$ is a connectivity-preserving mapping.
3. The assignment $\mu$ is a cycle-preserving mapping.

**Proof.** This consequence follows from Lemma 3.11, Lemma 3.14 and Lemma 3.16.

Based on the above condition we get the next theorem.

**Theorem 3.19 (Polynomial-Time Verification)** Deciding whether a given assignment $\mu$ of a query w.r.t. a database is a semiflexible matching is in polynomial time in the size of the query and the database.

In summary, Theorem 3.19 states that for all types of queries, testing whether $\mu$ is a semiflexible matching has a running time that is polynomial in the size of the query and the database.

### 3.2 Query Evaluation in Semiflexible Semantics

In *data complexity*, query evaluation is measured in terms of the size of the database. That is, we assume that the size of the query is fixed. Query evaluation has a polynomial-time data complexity under either the semiflexible or flexible semantics.

In *combined complexity*, both the query and the data are considered as input. The combined complexity of query evaluation under either the semiflexible and the
flexible semantics is exponential, since the size of the result could be exponential in the size of the query and the data.

A better approach is to analyze the input-output complexity that is measured in terms of the size of the query, the database and the result. The motivation for using input-output complexity is that it allows us to distinguish between the following two cases. First, a case where the runtime of the algorithm is exponential simply because the size of the result is exponential. Secondly, a case where the runtime of the algorithm is exponential even though the size of the result is small (i.e., less than exponential in the size of the input). The second case could be a consequence of having a naive algorithm, i.e., an algorithm that does a lot of redundant work. A different cause could be that the problem is NP-hard.

Recall that in the relational case, merely checking whether a join of \( n \) relations is not empty is NP-complete [49]; hence, the input-output complexity is exponential. However, for the important case of acyclic joins, the input-output complexity is polynomial [65]. In this section, we will discuss the input-output complexity of query evaluation under the semiflexible semantics.

### 3.2.1 Path Queries

In this section we consider path queries. A path query has the form \( v_0l_1v_1l_2v_2\ldots l_nv_n \), where \( v_0 \) is the root and all the \( v_i \) are distinct. First, we will discuss the case of evaluating a path query over a database that is also a path. Note that even in this case, the result could be exponential in the size of the query and the database (provided that some labels are repeated along paths of the database). However, the input-output complexity is polynomial.

For a query node \( v \), the correspondence set of \( v \), denoted \( C_v \), is the set of all database objects \( o \), such that \( o \) satisfies some woc of \( v \) (i.e., there is a label \( l \), such that both \( o \) and \( v \) have incoming edges labeled with \( l \)). The correspondence set of the root \( r_Q \) of \( Q \) consists of the root \( r_D \) of the database.
Proposition 3.20 (Path Query and Path Database) Let $Q$ be a path query of the form $v_0l_1v_1l_2v_2 \ldots l_nv_n$, and $D$ be a path database over a set of objects $O$. Consider the set $\mathcal{M}$ that the next equation defines.

$$
\mathcal{M} = \{ \mu : V \to O \mid \mu(v_i) \in C_{v_i} \text{ for } 0 \leq i \leq n, \text{ and } \mu(v_i) \neq \mu(v_j) \text{ for } i \neq j \}. \quad (3.1)
$$

Then the set $\mathcal{M}_{sf}(Q, D)$ of the semiflexible matchings of $Q$ w.r.t. $D$ is equal to $\mathcal{M}$. Computing $\mathcal{M}_{sf}(Q, D)$ has linear-time input-output complexity.

Proof. First, we show that $\mathcal{M} = \mathcal{M}_{sf}(Q, D)$. Consider a mapping $\mu$ in $\mathcal{M}$. Obviously, $\mu$ satisfies the root constraint because the correspondence set of the root, $C_{v_0}$, contains only the database root. In addition, $\mu$ maps the variables of $Q$ into a database path. Thus, $\mu$ is a connectivity-preserving mapping (see Definition 3.10) and according to Corollary 3.13, $\mu$ is a semiflexible matching. For the other direction, suppose that $\mu$ is in $\mathcal{M}_{sf}(Q, D)$. The single path in $Q$ satisfies the SF-Condition w.r.t. $\mu$. Thus, for each query variable $v_i$, $\mu(v_i) \in C_{v_i}$. Furthermore, $\mu(v_i) \neq \mu(v_j)$ for every $i \neq j$. Consequently, $\mu$ satisfies the condition in Equation 3.1, and hence, $\mu$ is in $\mathcal{M}$.

To show that the computation can be done in linear time, let $|Q|$, $|D|$ and $|M|$ be the sizes of the query, the database and the result, respectively.

Computing $\mathcal{M}_{sf}(Q, D)$ is done as follows. First, we verify that for each label $l$ in $Q$ if $l$ is attached to $k$ edges of $Q$, then there are at least $k$ different nodes in $D$ with an incoming edge labeled with $l$. If this test fails, then $\mathcal{M}_{sf}(Q, D)$ is empty. The test requires a runtime of $O(|Q| + |D|)$, since counting the labels can be done in a single pass over $Q$ and $D$.

Next, the correspondence sets $C_{v_i} \ (1 \leq i \leq n)$ are constructed. Using a suitable data structure, such as a hash table, this construction can be done in $O(|Q| + |D|)$.

Finally, the matchings are created. A matching is constructed by choosing a single object from each correspondence set, verifying that no object is chosen twice. This can be done using nested iterations over the sets $C_{v_i} \ (1 \leq i \leq n)$. Constructing $\mathcal{M}$ from the correspondence sets requires $O(|M|)$ runtime. Thus, the runtime of the algorithm is in $O(|Q| + |D| + |M|)$. 

\[ \square \]
Chapter 3  Flexible and Semiflexible Queries

Next, we investigate the case where the database is a tree. A simple evaluation algorithm for a path query $Q$ w.r.t. a tree database $D$ would be as follows. For each path $\pi$ in $D$, from the root to a leaf, evaluate $Q$ w.r.t. $\pi$. Consider the sets of semiflexible matchings that are the result of evaluating $Q$ w.r.t. the different paths. The union of these sets is the result of evaluating $Q$ w.r.t. the whole database. Evaluating $Q$ w.r.t. the paths of $D$ can be done as described in Proposition 3.20.

The runtime complexity of such evaluation will be a function of $|D|^2$. This is because the number of paths in $D$, from the root to a leaf, could be $O(|D|)$. In addition, the size of such paths could be $O(|D|)$.

The simple evaluation algorithm that was described in the last paragraphs could be improved so that the runtime will not be a function of $|D|^2$. The algorithm is presented in Appendix C.

**Theorem 3.21 (Path Query and Tree Database)** Let $Q$ be a path query and $D$ be a tree database. There is an algorithm that computes the semiflexible matchings of $Q$ w.r.t. $D$ in $O(|Q||D| + |M|)$ runtime, where $|Q|$, $|D|$ and $|M|$ are the sizes of the query, the database and the result, respectively.

The proof of Theorem 3.21 is presented in Appendix C.

Next, we show that query evaluation, under the semiflexible semantics, is not likely to have a polynomial-time input-output complexity when the database is a dag. We use a reduction of 3SAT. A formula $\varphi$ is in 3CNF if $\varphi$ is a conjunction of clauses $c_1, \ldots, c_m$, where each clause $c_i$ is a disjunction of three literals $l_{i1}, l_{i2}, l_{i3}$ and a literal is either a propositional letter or a negation of a propositional letter. Deciding whether a formula in 3CNF has a satisfying assignment is NP-complete [35].

**Lemma 3.22 (Reduction of 3SAT)** Consider a 3CNF formula $\varphi$ over a set of propositional letters $U$. One can construct in polynomial time a path query $Q$ and a dag database $D$ such that the following are equivalent:

- There is a semiflexible matching of $Q$ w.r.t. $D$.
- There is an assignment for the propositional letters in $U$ that satisfies $\varphi$. 
Chapter 3  Flexible and semiflexible Queries

The proof of Lemma 3.22 is given in Appendix C.

**Theorem 3.23 (NP-Completeness)** Given a path query $Q$ and a dag database $D$, deciding whether $\mathcal{M}_{sf}(Q, D)$ is not empty is NP-complete.

**Proof.** NP-hardness follows from Lemma 3.22. Membership in NP is because one can guess in polynomial time a mapping from the variables of $Q$ to the objects of $D$ and verify, by Corollary 3.13, that this mapping is a semiflexible matching.  

**Proposition 3.24 (Path Query and Cyclic Database)** Given a path query $Q$ and a cyclic database $D$, query evaluation has an $O(|D|^{|Q|-1}|Q|^2)$ runtime under input-output complexity.

**Proof.** As a preliminary step, we compute a table that holds an entry for each pair of nodes $o_1$ and $o_2$ in $D$. The entry for $o_1$ and $o_2$ contains the set of labels $l$ such that $D$ has a path of the form $o_1 * l o_2$. If there is no path in $D$ from $o_1$ to $o_2$, then the entry contains an empty set. In addition, we topologically sort the nodes of $D$ and compute the strongly connected components of $D$ in $O(|D|)$ (see in [27] how to compute strongly connected components in linear time). All together, the preliminary step is done in $O(|D|^2)$.

Next, we iteratively test all the possible assignments. For each assignment it is checked whether it is a semiflexible matching. Since the root of the query is always mapped to the root of the database, there are $|D|^{|Q|-1}$ possible mappings to check. Verifying that an assignment is a semiflexible matching is done as described in Corollary 3.13. According to Corollary 3.13, it should be checked that the assignment is a connectivity-preserving mapping (see Definition 3.10); and this requires a test w.r.t. each pair of query variables, i.e., $O(|Q|^2)$ tests.  

**3.2.2 Tree and DAG Queries over a Tree Database**

XML documents are frequently trees. Thus, querying tree databases has a particular importance. In this section, we show that, over a tree database, evaluation of tree
and dag queries can be done in polynomial time, in the size of the input and the output, provided that labels are not repeated in query paths.

When both the query and the database are acyclic, queries that have repeated label on a path are not practical, for the following reasons. Acyclic databases seldom have the same label repeated on a path, since it may cause semantic confusion. In the rare cases, when the database does have repeated label on a path, the semantic meaning of objects with the same incoming label is defined by the order of their appearance on the path. In semiflexible matchings, however, this order is not preserved.

We say that a query is \textit{plain} if there are no repeated labels on query paths. Consider a plain tree query and a tree database. In the first part of this section, we will present an algorithm that computes the semiflexible matchings of the query w.r.t. the database, in polynomial time in the size of the input and output. In the second part of this section, we will show how to modify the algorithm in order to compute, in polynomial time, semiflexible matchings of a simple dag query w.r.t. the database. We start by presenting some definitions and notations.

\textbf{Preliminaries}

Suppose that a tree or a dag query is posed to a tree database. If there is a variable in the query that has two different incoming labels (on two different edges), then there are no semiflexible matchings of the query w.r.t. the database. This is because, in a tree database, an object (other than the root) has exactly one incoming edge and a single incoming label. Thus, we will consider queries in which the mapping of nodes to their incoming labels is a function, i.e., for each node, all the edges that enter the node have the same label. We denote by \textit{label}(v) the label on the edges that enter v. The same notation will also be used for the mapping of objects to labels in a tree database. We assume that for the root of the query and for the root of the database, the \textit{label()} function returns the unique label \textit{ROOT}.

In a tree database $D$, the topological order of the nodes is a partial order that can be used to distinguish between pairs of nodes that are connected by a path and pairs of nodes that are not connected by a path. We say that a node $o_1$ is \textit{above} $o_2$
and $o_2$ is below $o_1$, denoted $o_1 \triangleright o_2$, if there is a path in $D$ from $o_1$ to $o_2$.\footnote{In some places we use the terms “ancestor” and “descendent” instead of the terms “above” and “below”. By definition, $o_1$ is an ancestor (descendent) of $o_2$ when $o_1$ is above (below) $o_2$.} By $o_1 \triangleright o_2$ we denote the case where $o_1 \triangleright o_2$ or $o_1 = o_2$. Note that $r_D \triangleright o$ for the root of the database $r_D$ and any object $o$. Moreover, if $o \not= r_D$ then $r_D \triangleright o$. Two nodes $o_1$ and $o_2$ are on a path in $D$ if $o_1$ is either above $o_2$ or below $o_2$. For acyclic queries, “above” and “below” are defined similarly.

Consider a set of objects $O$ in a tree database $D$. Suppose that $O'$ are all the objects in $D$ that are on a path with each object of $O$. The following lemma shows how to find $O'$.

**Lemma 3.25** Let $D$ be a tree database.

1. Let $o_1, \ldots, o_k$ be objects that are all on one path in $D$ and let $o_\downarrow$ be the lowest node among $o_1, \ldots, o_k$, i.e., the object that is the farthest from the root. Then an object $o$ in $D$ is on one path with all the nodes $o_1, \ldots, o_k$ if and only if \((o \preceq o_i) \lor (o \succeq o_i)\), i.e., $o$ is in the set \(\{x \mid (x \preceq o_i) \lor (x \succeq o_i)\}\).

2. Let $o_1, \ldots, o_k$, where $k \geq 2$, be objects such that no pair, among these objects, is connected by a path. Let $o_\uparrow$ be the lowest common ancestor of $o_1, \ldots, o_k$. Then, an object $o$ of $D$ is connected by a path to each one of the objects $o_1, \ldots, o_k$ if and only if $o \succeq o_\uparrow$, i.e., $o$ is in the set \(\{x \mid x \succeq o_\uparrow\}\).

**Proof.** We start with Part 1 of the lemma. If $o$ is below $o_\downarrow$ (i.e., $o \preceq o_\downarrow$) then for each $1 \leq i \leq k$ there is a path from $o_i$ to $o_\downarrow$ and a path from $o_\downarrow$ to $o$. Thus, there is a path to $o$ from each one of the $k$ objects. If $o$ is above $o_\downarrow$ (i.e., $o \succeq o_\downarrow$) then there is a path from $o$ to $o_\downarrow$. If there was a node among $o_1, \ldots, o_k$ that did not have a path to $o$ and was not reachable by a path from $o$, then there would be two different paths in $D$ from the root to $o_\downarrow$. This would be a contradiction to the fact that $D$ is a tree. For the other direction, if $(o \not\preceq o_\downarrow) \land (o \not\succeq o_\downarrow)$ then there is no path from $o$ to $o_\downarrow$ and there is no path from $o_\downarrow$ to $o$. That is, $o$ is not on one path with $o_\downarrow$. (Recall that $o_\downarrow$ is one of the nodes $o_1, \ldots, o_k$.)

We now prove Part 2 of the lemma. If $o \succeq o_\uparrow$ then there is a path from $o$ to $o_\uparrow$. There is a path from $o_\uparrow$ to each one of the objects $o_1, \ldots, o_k$, since $o_\uparrow$ is an ancestor
Chapter 3  Flexible and Semiflexible Queries

of all these objects. So, $o$ is on a path with each one of the $k$ objects. For the other
direction, assume that $o \not\preceq o_\uparrow$. There are three possible cases to consider. The first
case is that $o$ is below $o_\uparrow$ and there is a path from $o$ to all the objects $o_1, \ldots, o_k$.
This would be a contradiction to $o_\uparrow$ being the least common ancestor. The second
case is that $o$ is below $o_\uparrow$ and it is also below the object $o_i$, for some $1 \leq i \leq k$. In
this case, there cannot be an $o_j$, $j \neq i$, such that $o$ is also below $o_j$, because $D$ is a
tree and there is no path between $o_i$ and $o_j$. So, $o$ is below $o_i$ and $o_j$ is below $o$. This
contradicts the condition that no pair of objects among $o_1, \ldots, o_k$ are connected by
a path. The third case is when $o$ is not on the same path with $o_\uparrow$. In this case, there
is no path from any of the objects $o_1, \ldots, o_k$ to $o$. There cannot be a path from $o$ to
any of the nodes $o_1, \ldots, o_k$, because $D$ is a tree and there is exactly one path from
the root to each node. ❄

The partial ordering $\succ$ is used for defining SF-properties.

Definition 3.26 (SF-Property) Suppose that $Q$ is a query and $D$ is a database.
A matching property w.r.t. $Q$ and $D$ is a 4-tuple $P = (v, o, V_P, S_P)$ where $v$ is a
variable of $Q$, $o$ is an object of $D$, $V_P$ is a set of variables of $Q$ and $S_P$ is a set of
objects of $D$. The matching property $P$ is an SF-property if the following holds. For
every semiflexible matching $\mu$ of $Q$ w.r.t. $D$, if $\mu$ maps $v$ to $o$, then $\mu$ maps each
variable of $V_P$ to an object in the set $S_P$.

Later in this section, we will present an algorithm that computes semiflexible
matchings of a tree query w.r.t. a tree database. In the algorithm, there are two
phases. In the first phase, the algorithm computes SF-properties w.r.t. the given
query and database. The SF-properties are expressed using the partial ordering $\succ$.
This is demonstrated in the following two examples.

Example 3.27 Consider a query $Q$ and a variable $v$ in $Q$, suppose that $v$ is mapped
to a database node $o$ by a semiflexible matching $\mu$. Then, $\mu$ maps all the ancestors
of $v$ to nodes that are either above $o$ or below $o$ in the database, i.e., to nodes in the
set $\{ x \mid (x \succ o) \lor (x \prec o) \}$.
Example 3.28 Consider a variable $v$ that has two children $v_1$ and $v_2$ in a query $Q$. Suppose that $v_1$ and $v_2$ are mapped by a semiflexible matching $\mu$ to the objects $o_1$ and $o_2$, respectively, where $o_1 \nleq o_2$ and $o_2 \nleq o_1$. Let $o$ be the least common ancestor of $o_1$ and $o_2$ in the database. Then $\mu$ maps $v$ to an object in the set $\{x \mid x \succ o\}$.

SF-properties are defined by abstract mappings. Note that the following definition of an abstract mapping is recursive.

Definition 3.29 (Abstract Mapping) Let $Q$ be an acyclic query, $D$ be a tree database and $v$ be a variable in $Q$. Consider a pair $(o, S)$, where $o$ is an object of $D$ and $S$ is a set of objects of $D$. The pair $(o, S)$ is an abstract mapping of $v$ w.r.t. $D$ if the following three conditions hold.

1. $\text{label}(v) = \text{label}(o)$.

2. All the objects in $S$ are on a path with $o$.

3. Suppose that $v$ has children $v_1, \ldots, v_k$ in $Q$. Then there are abstract mappings $(o_1, S_1), \ldots, (o_k, S_k)$ of $v_1, \ldots, v_k$, respectively, such that, for each $1 \leq i \leq k$, $S \subseteq S_i$, and $o \in S_i$.

Proposition 3.30 (Completeness) Suppose that $Q$ is an acyclic query and $D$ is a tree database. Let $\mu$ be a semiflexible matching of $Q$ w.r.t. $D$ and let $v$ be a variable in $Q$. Then, $(\mu(v), S)$, where $S = \{\mu(u) \mid u \succ v\}$, is an abstract mapping of $v$ w.r.t. $D$.

Proof. We need to show that $(\mu(v), S)$ satisfies the three conditions of Definition 3.29. According to Corollary 3.13, $\mu$ is a connectivity-preserving mapping (Definition 3.10) and, thus, Condition 1 and Condition 2 are satisfied.

Satisfaction of Condition 3 is shown by induction on the structure of $Q$. If $v$ is a leaf, then Condition 3 holds in a trivial way since $v$ has no children. The inductive hypothesis is that $(\mu(v'), \{\mu(u) \mid u \succ v'\})$ satisfies Condition 3 of Definition 3.29, for every variable $v'$ that has less than $n$ descendents.

Consider a variable $v$ that has $n$ descendents in $Q$. Let $v_1, \ldots, v_k$ be the children of $v$. For each $1 \leq i \leq k$, the variable $v_i$ has less then $n$ descendents. Thus, according
to the inductive hypothesis, \((\mu(v_i), S_i)\), where \(S_i = \{\mu(u) \mid u \succ v_i\}\), is an abstract mapping of \(v_i\) w.r.t. \(D\). Then, for each \(1 \leq i \leq k\), \(S \subseteq S_i\) and \(\mu(v) \in S_i\). Hence, \((\mu(v), \{\mu(u) \mid u \succ v\})\) satisfies Condition 3 of Definition 3.29.

Consider an acyclic query \(Q\), a tree database \(D\) and a variable \(v\) of \(Q\). Let \(o\) be an object in \(D\) such that \(\text{label}(o) = \text{label}(v)\). We say that an abstract mapping \((o, S)\) of \(v\) is maximal if \(S' \subseteq S\) holds for every abstract matching \((o, S')\) of \(v\). The goal of the next proposition is to show that if a variable \(v\) has an abstract matching then \(v\) also has a maximal abstract matching.

**Proposition 3.31 (Applying Union)** Suppose that \(Q\) is an acyclic query, \(D\) is a tree database and \(v\) is a variable of \(Q\). If \((o, S^1)\) and \((o, S^2)\) are two abstract mappings of \(v\), then also \((o, S^1 \cup S^2)\) is an abstract mapping of \(v\).

**Proof.** Because \((o, S^1)\) and \((o, S^2)\) are abstract mappings of \(v\), Condition 1 in Definition 3.29 is satisfied. Condition 2 is satisfied w.r.t. \(S^1\) and \(S^2\). Thus, it is satisfied w.r.t. \(S^1 \cup S^2\).

We complete the proof of the proposition by induction on the structure of \(Q\). If \(v\) is a leaf then Condition 3 is satisfied in a trivial way, i.e., \(v\) has no children. Thus the proposition holds for \(v\).

The inductive hypothesis is that the proposition holds for all the nodes \(v\) in \(Q\) that have a height of at most \(n - 1\). In the following we show that if the inductive hypothesis is true then the proposition holds for nodes that have a height of \(n\).

Consider a node \(v\) in \(Q\) that has a height of \(n\) and let \((o, S^1)\) and \((o, S^2)\) be abstract mappings of \(v\). We need to show that Condition 3 in Definition 3.29 is satisfied w.r.t. \((o, S^1 \cup S^2)\).

Suppose that \(v_1, \ldots, v_k\) are the children of \(v\). Since \((o, S^j)\) \((j = 1, 2)\) is an abstract mapping of \(v\), there are abstract mappings \((o, S^i_1), \ldots, (o, S^i_k)\) of \(v_1, \ldots, v_k\), respectively, such that \(S^j_i \subseteq S^i_j\), for each \(1 \leq i \leq k\). By the inductive hypothesis, \((o, S^i_1 \cup S^i_2), \ldots, (o, S^i_k \cup S^i_k)\) are abstract mappings of \(v_1, \ldots, v_k\), respectively. These abstract mappings show that Condition 3 is satisfied w.r.t. \((o, S^1 \cup S^2)\).

Suppose that \((o, S_1), \ldots, (o, S_m)\) are all the abstract mappings of \(v\) in which the first element is \(o\). Let \(S^{max} = \bigcup_{i=1}^{m} S_i\). Then, from Proposition 3.31, it follows that
Chapter 3  Flexible and Semiflexible Queries

(o, S^{max}) is an abstract mapping of v. Furthermore, S_i \subseteq S^{max}, for each 1 \leq i \leq m. Thus, S^{max} is a maximal abstract mapping of v.

Next, we show that a maximal abstract mapping (o, S) of v defines an SF-property P = (v, o, \{x \mid x \succ v\}, S) w.r.t. Q and D. Note that \{x \mid x \succ v\} are the ancestors of v in Q.

**Proposition 3.32 (Abstract Mappings Define SF-Properties)** Suppose that Q is an acyclic query, D is a tree database and v is a variable of Q. If (o, S) is a maximal abstract mapping of v w.r.t. D then P = (v, o, \{x \mid x \succ v\}, S) is an SF-property w.r.t. Q and D.

**Proof.** We need to show that P is an SF-property w.r.t. Q and D. That is, if a semiflexible matching of Q w.r.t. D maps v to o then it also maps each ancestor of v to an object of S.

Consider a semiflexible matchings \mu of Q w.r.t. D. Suppose that \mu maps v to o and \hat{v} \succ v, i.e., \hat{v} is a variable above v. Then, we need to show that \mu(\hat{v}) is in S.

Consider the pair (\mu(v), S_v), where S_v = \{\mu(w) \mid w \succ v\}). According to Proposition 3.30, (\mu(v), S_v) is an abstract mapping of v. In addition, S_v contains \mu(\hat{v}), because \hat{v} is above v. Since (o, S) is a maximal abstract mapping w.r.t. v and \mu(v) = o, it holds that S_v \subseteq S. Hence, \mu(\hat{v}) \in S. \hfill \Box

**Computing Abstract Mappings**

Now, we will show how to compute abstract mappings in polynomial time in the size of the query and the database. The next proposition is a first step towards computing abstract mappings.

**Proposition 3.33 (Bottom-Up Evaluation)** Suppose that Q is an acyclic query, D is a tree database, v is a variable of Q and v_1, \ldots, v_k are the children of v (if there are any). Then (o, S) is an abstract mapping of v if and only if the following two conditions hold.

1. label(o) = label(v).
2. There are abstract mappings \((o_1, S_1), \ldots, (o_k, S_k)\) of \(v_1, \ldots, v_k\), respectively, such that

(a) \(o \in S_i\), for each \(1 \leq i \leq k\), and

(b) \(S \subseteq S_1 \cap S_2 \cap \cdots S_k \cap \{x \mid x \leq o \lor x \geq o\}\). (If \(v\) is a leaf of \(Q\), then \(S \subseteq \{x \mid x \leq o \lor x \geq o\}\).)

**Proof.** Suppose that Conditions 1 and 2 of the proposition hold. We need to show that \((o, S)\) is an abstract mapping. We show that the three conditions in Definition 3.29 hold. Condition 1 of Definition 3.29 is the same as Condition 1 in the proposition. Thus, it holds.

We show that Condition 2 of Definition 3.29 holds. The set \(S\) is contained in the set \(S_p = \{x \mid x \leq o \lor x \geq o\}\), because \(S\) is the intersection of \(S_p\) with other sets. The set \(S_p\) is the set of objects that are on a path with \(o\). Thus, all the objects in \(S\) are on a path with \(o\), which is what we needed to show. To prove that Condition 3 of Definition 3.29 holds, we need to show that \(S \subseteq S_i\), for each \(1 \leq i \leq k\). Since \(S\) is an intersection of \(S_i\) with other sets, obviously, \(S \subseteq S_i\). From Condition 2 it follows that \(o \in S_i\), for each \(1 \leq i \leq k\).

For the other direction, suppose that \((o, S)\) is an abstract mapping of \(v\). We need to show that Conditions 1 and 2 of the proposition hold. Condition 1 of the proposition is equal to Condition 1 in Definition 3.29. Thus, it holds.

We claim that Condition 2 of the proposition follows from Condition 2 and Condition 3 of Definition 3.29. Since \((o, S)\) is an abstract mapping w.r.t. \(v\), there are abstract mappings \((o_1, S_1), \ldots, (o_k, S_k)\) of \(v_1, \ldots, v_k\), respectively, such that for each \(1 \leq i \leq k\), \(S \subseteq S_i\) and \(o \in S_i\). Thus, \(S \subseteq S_1 \cap S_2 \cap \cdots S_k\). The set \(S_p = \{x \mid x \leq o \lor x \geq o\}\) contains all the objects that are on a path with \(o\). Thus, because of Condition 2 in Definition 3.29, \(S \subseteq S_p\). That is, \(S \subseteq S_1 \cap S_2 \cap \cdots S_k \cap S_p\). This proves our claim. \(\square\)

**Corollary 3.34 (Maximal Elements)** In Proposition 3.33, if \((o, S)\) is a maximal abstract mapping of \(v\) then there are \((o_1, S_1), \ldots, (o_k, S_k)\) that satisfy the conditions of the proposition and are maximal abstract mappings of \(v_1, \ldots, v_k\), respectively.
addition, the equality $S = S_1 \cap S_2 \cap \cdots S_k \cap \{x \mid x \leq o \lor x \geq o\}$ holds (if $v$ is a leaf, $S = \{x \mid x \leq o \lor x \geq o\}$).

Proof. If $v$ is a leaf then $(o, \{x \mid x \leq o \lor x \geq o\})$ is a maximal abstract mapping of $v$ and the conditions of Proposition 3.33 are satisfied. Suppose that $v$ is not a leaf.

Let $v_1, \ldots, v_k$ be the children of $v$. Consider a maximal abstract mapping $(o, S)$. According to Proposition 3.33, there are abstract mappings $(o_1, S_1), \ldots, (o_k, S_k)$ of $v_1, \ldots, v_k$, respectively, such that $o \in S_i$, for each $1 \leq i \leq k$, and $S_1 \subseteq S_1 \cap S_2 \cap \cdots S_k \cap \{x \mid x \leq o \lor x \geq o\}$.

Let $(o_1, S_1^m), \ldots, (o_k, S_k^m)$ be maximal abstract mappings. Let $S^m = S_1^m \cap S_2^m \cap \cdots S_k^m \cap \{x \mid x \leq o \lor x \geq o\}$. Then, $S_i \subseteq S_i^m$, for each $1 \leq i \leq k$ and, hence, $S \subseteq S^m$. According to Proposition 3.33, $(o, S^m)$ is an abstract mapping of $v$. Because $(o, S)$ is a maximal abstract mapping of $v$, it holds that $S \supseteq S^m$. Therefore, $S = S^m$, i.e., $S$ is the intersection of sets in maximal abstract mappings.

Consider an abstract mapping $(o, S)$ w.r.t. some variable $v$ in a query. The set $S$ has $O(|D|)$ size, where $D$ is the size of the database. However, we will show that it is not necessary to actually compute $S$. Instead, for each set $S$ in an abstract mapping, we compute an abstract set which is a notation of a definition of $S$. Such a notation is essentially a constraint such that $S$ contains all the database objects that satisfy this constraint.

Definition 3.35 (Abstract Sets) Let $o$ be a database object. A path abstract set (pas) is a set notation of the form $\{x \mid x \geq o\}$. A tree abstract set (tas) is a set notation of the form $\{x \mid x \leq o \lor x \geq o\}$. A basic abstract set is a set notation that is either a pas or a tas.

Two abstract sets are equivalent (denoted $\equiv$) if the sets that they represent are equal. The following proposition shows that over a tree database, every conjunction of basic abstract sets can be written as an equivalent basic abstract set.

Proposition 3.36 (Equivalence Rules for Conjunction) The following equivalences hold in a tree database $D$. 
Procedure \textit{combine}(S_1, S_2);

\textbf{Input} Two basic abstract sets $S_1$ and $S_2$;

\textbf{Output} A basic abstract set that is equivalent to $S_1 \cap S_2$;

if $S_1$ is a pas and $S_2$ is a tas then swap $S_1$ and $S_2$;

let $o_1$ and $o_2$ be the objects that appear in $S_1$ and $S_2$, respectively;

if $o_1 \succeq o_2$ or $o_2 \succeq o_1$ then

let $o_h$ be the highest of $o_1$ and $o_2$ ($o_h = o_1$ if $o_1 = o_2$);

let $o_l$ be the lowest of $o_1$ and $o_2$ ($o_l = o_2$ if $o_1 = o_2$);

\textbf{case} $S_1 = \{x \mid x \preceq o_1 \lor x \succeq o_1\}$ and $S_2 = \{x \mid x \preceq o_2 \lor x \succeq o_2\}$:

\[ \hat{S} = \{x \mid x \preceq o_h \lor x \succeq o_l\} \];

\textbf{case} $S_1 = \{x \mid x \succeq o_1\}$ and $S_2 = \{x \mid x \succeq o_2\}$:

\[ \hat{S} = \{x \mid x \succeq o_h\} \];

\textbf{case} $S_1 = \{x \mid x \preceq o_1 \lor x \succeq o_1\}$ and $S_2 = \{x \mid x \succeq o_h\}$:

\[ \hat{S} = \{x \mid x \preceq o_l\} \];

\textbf{case} $S_1 = \{x \mid x \preceq o_h \lor x \succeq o_h\}$ and $S_2 = \{x \mid x \succeq o_l\}$:

\[ \hat{S} = \{x \mid x \succeq o_l\} \];

else (* $o_1 \nprec o_2$ and $o_2 \nprec o_1$ *)

let $o_c$ be the least common ancestor of $o_1$ and $o_2$;

\[ \hat{S} = \{x \mid x \preceq o_c\} \]

return $\hat{S}$;

\begin{figure}[h]
\centering
\begin{tikzpicture}
\end{tikzpicture}
\caption{For a conjunction of two basic abstract sets, \textit{combine} computes an equivalent basic abstract set (without conjunction), according to the seven rules of Proposition 3.36.}
\end{figure}

\begin{enumerate}
\item If $o_2 \succeq o_1$ then

(a) $\{x \mid (x \preceq o_1 \lor x \succeq o_1) \land (x \preceq o_2 \lor x \succeq o_2)\} \equiv \{x \mid x \preceq o_1 \lor x \succeq o_1\}$

(b) $\{x \mid x \succeq o_1 \land x \succeq o_2\} \equiv \{x \mid x \succeq o_2\}$

(c) $\{x \mid (x \preceq o_1 \lor x \succeq o_1) \land x \preceq o_2\} \equiv \{x \mid x \preceq o_2\}$

(d) $\{x \mid (x \preceq o_2 \lor x \succeq o_2) \land x \preceq o_1\} \equiv \{x \mid x \preceq o_1\}$
\end{enumerate}
2. If $o$ is the least common ancestor of $o_1$ and $o_2$ and $(o_2 \not\preccurlyeq o_1) \land (o_2 \not\succeq o_1)$, i.e., $o_1$ and $o_2$ are not on the same path, then

   \[ \{ x \mid x \preceq o_1 \lor x \succeq o_2 \} \equiv \{ x \mid x \succeq o \} \]

   \[ \{ x \mid x \succeq o_1 \land x \succeq o_2 \} \equiv \{ x \mid x \succeq o \} \]

   \[ \{ x \mid x \succeq o_1 \land (x \preceq o_2 \lor x \succeq o_2) \} \equiv \{ x \mid x \succeq o \} \]

Proof. All the equivalences directly follow from the definition of the partial order $\succeq$ and the properties of $D$ as a tree. \qed

We use the equivalence rules of Proposition 3.36 to write a conjunction of basic abstract sets as a basic abstract set. A function \textit{combine} that applies the rules of Proposition 3.36 to a conjunction of two basic abstract sets is presented in Figure 3.2.

Our goal is to compute maximal abstract matchings. This requires discarding the non-maximal elements from the sets of abstract matchings that the algorithm computes.

Let $D$ be a tree database. We say that an abstract set $S_2$ contains an abstract set $S_1$ w.r.t. $D$, denoted $S_1 \subseteq S_2$, if the set of objects that $S_1$ defines is contained in the set of objects that $S_2$ defines. The following proposition presents containment rules.

**Proposition 3.37 (Containment Rules)** Let $D$ be a tree database and let $o_1$ and $o_2$ be objects in $D$ such that $o_2 \succeq o_1$. The following containment rules hold.

1. $\{ x \mid x \succeq o_1 \} \subseteq \{ x \mid x \preceq o_2 \lor x \succeq o_2 \}$

2. $\{ x \mid x \preceq o_1 \lor x \succeq o_1 \} \subseteq \{ x \mid x \preceq o_2 \lor x \succeq o_2 \}$

3. $\{ x \mid x \succeq o_2 \} \subseteq \{ x \mid x \succeq o_1 \}$

4. $\{ x \mid x \succeq o_2 \} \subseteq \{ x \mid x \preceq o_1 \lor x \succeq o_1 \}$

Proof. All the containment rules directly follow from the definition of the partial order $\succeq$ and the properties of $D$ as a tree. \qed

The following proposition shows that the set of containment rules in Proposition 3.37 is complete.
Proposition 3.38 (Containment Rules, Completeness) Consider a tree database $D$. Let $S_1$ and $S_2$ be basic abstract sets such that $S_1 \subseteq S_2$. Let $o_1$ and $o_2$ be the objects that appear in the definitions of $S_1$ and $S_2$, respectively. Then, one of the following two cases is true.

- It holds that $o_2 \preceq o_1$ and the containment $S_1 \subseteq S_2$ has the form of Rule 1 or Rule 2 in Proposition 3.37.

- It holds that $o_1 \preceq o_2$ and the containment $S_1 \subseteq S_2$ has the form of Rule 3 or Rule 4 in Proposition 3.37.

Proof. If $o_2 \not\preceq o_1$ and $o_1 \not\preceq o_2$, then the containment $S_1 \subseteq S_2$ does not hold. When $o_2 \succeq o_1$ or $o_1 \succeq o_2$, a simple case analysis proves the claim. □

In Figure 3.3, a procedure remove-non-maximal is presented. The procedure receives a database $D$ and a set $A$ of abstract mappings of a variable $v$ w.r.t. $D$. In the procedure, if $A$ contains two abstract matchings $(o, S_1)$ and $(o, S_2)$ such that $S_1 \subseteq S_2$, according to one of the rules in Proposition 3.37, then $(o, S_1)$ is removed from $A$.

Computing Abstract Mappings

Now, we show how to compute the maximal abstract mappings of variables in an acyclic query $Q$ w.r.t. a tree database $D$. Procedure MAMA computes the set of maximal abstract mappings of each variable in $Q$. The computation is performed in a bottom-up fashion, starting with the leaves of the query and rising towards the root (i.e., an order that is inverted to the topological order of $Q$). For each node $v$, MAMA computes a set $A_v$ that consists of all the maximal abstract mappings of $v$.

In the following, we describe how to compute the sets $A_v$. The computation is divided into several cases. The cases are according to the the number of children that $v$ has.

Case 1 Suppose that $v$ is a leaf of $Q$. Then, for each object $o$ in $D$ such that $\text{label}(o) = \text{label}(v)$, the abstract mapping $(o, \{x \mid x \preceq o \lor x \succeq o\})$ is added to $A_v$. Note that there are no non-maximal elements that should be removed.
Procedure \texttt{remove-non-maximal}(A, D);

\textbf{Input} A database \(D\) and a set of abstract mappings \(A\)
of a variable \(v\) w.r.t. \(D\);

\textbf{Output} The set \(A\) after removing abstract matchings \((o, S)\) for which
there is another abstract matching \((o, S')\) in \(A\) such that \(S \subseteq S'\).

for each two abstract mappings \((o_1, S_1)\) and \((o_2, S_2)\) in \(A\) such that \(o_1 = o_2\)
let \(o_{c_1}\) and \(o_{c_2}\) be the objects in \(S_1\) and \(S_2\), respectively;
if \(o_{c_1} \preceq o_{c_2}\) then
  if \(S_2\) has the form \(\{x \mid x \preceq o_{c_2} \lor x \succeq o_{c_2}\}\) then
    remove \((o_1, S_1)\) from \(A\);
  else if \(S_2\) has the form \(\{x \mid x \succeq o_{c_2}\}\) then
    remove \((o_2, S_2)\) from \(A\);
else if \(o_{c_2} \prec o_{c_1}\) then
  if \(S_1\) has the form \(\{x \mid x \preceq o_{c_1} \lor x \succeq o_{c_1}\}\) then
    remove \((o_2, S_2)\) from \(A\);
  else if \(S_1\) has the form \(\{x \mid x \succeq o_{c_1}\}\) then
    remove \((o_1, S_1)\) from \(A\);

Figure 3.3: The procedure receives a set of abstract mappings and removes abstract mappings that are not maximal. The removal is according to the four containment rules of Proposition 3.37.

\textbf{Case 2} Suppose that \(v\) has exactly one child \(v_1\) and \(A_{v_1}\) has already been computed. Initially, the algorithm finds all the objects \(o\) in \(D\) such that \(\text{label}(o) = \text{label}(v)\). If there is a pair \((o_1, S_1)\) in \(A_{v_1}\), such that \(o \in S_1\) and \(\text{label}(o) = \text{label}(v)\), then the abstract mapping \((o, \text{combine}(S_1, S_p))\), where \(S_p = \{x \mid x \preceq o \lor x \succeq o\}\), is added to \(A_v\). By executing \texttt{remove-non-maximal}(\(A_v, D\)), the non-maximal elements of \(A_v\) are removed.

\textbf{Case 3} Suppose that \(v\) has two or more children \(v_1, \ldots, v_k\) and \(A_{v_1}, \ldots, A_{v_k}\) have already been computed. The computation of \(A_v\) is performed in \(k\) steps.

In the first step, a set \(A'_1\) is computed from \(A_{v_1}\) in exactly the same way as the
computation of $A_v$ from $A_{v_1}$ in Case 2. In the $i$th step ($2 \leq i \leq k$), $A'_i$ is computed from $A'_{i-1}$ and $A_v$, as described next.

1. Iteratively, all the elements $(o, S)$ of $A'_{i-1}$ are considered.

2. For each $(o_i, S_i)$ in $A_{v_i}$ such that $o \in S_i$ and $\text{label}(o) = \text{label}(v)$, the abstract mapping $(o, \text{combine}(S, S_i))$ is added to $A'_i$.

3. By executing $\text{remove-non-maximal}(A'_i, D)$, the non-maximal elements of $A'_i$ are removed.

The set $A_v$ is the set $A'_k$.

**Proposition 3.39 (Correctness)** Consider a dag query and a tree database. For each variable $v$ of the query, the procedure MAMA computes the set of maximal abstract mappings of $v$ w.r.t. $D$.

**Proof.** Consider a variable $v$ and the set $A_v$ that was computed by MAMA for the variable $v$. Corollary 3.34 shows that the set $A_v$ contains all the maximal abstract mappings of $v$. Proposition 3.33 shows that all the elements of $A_v$ are abstract mappings. Finally, since non-maximal elements are removed, $A_v$ contains only maximal abstract mappings. □

**Proposition 3.40 (Complexity)** Consider a dag query $Q$ and a tree database $D$. Let $q$ and $d$ be the number of nodes in $Q$ and $D$, respectively. Each set $A_v$ that is computed by MAMA has at most $d$ elements. The time complexity of MAMA is in $O(q^2d^2)$.

**Proof.** The set $A_v$ consists of maximal abstract mappings of $v$. For an object $o$ of $D$, there can be at most one maximal abstract mapping of $v$. Thus, the size of $A_v$ cannot exceed the number of objects in $D$.

Now, we analyze the time complexity of MAMA. First, note that in $O(d^2)$ it is possible to construct a data structure that can tell in $O(1)$ whether two database objects are on a path. This is required for testing in $O(1)$ if a given object is an
element of an abstract set, i.e., an element of the concrete set that the abstract set represents.

Consider a set $A$ with $x$ abstract mappings. Removing the non-maximal elements from $A$ with the procedure remove-non-maximal is in $O(x)$ runtime. This is because, in one pass over $A$ the elements of $A$ can be inserted into a hash table, where the search keys of the hash table are the objects of the abstract mappings. Then, in each step, a pair of abstract mappings $(o, S_1)$, $(o, S_2)$ are picked. For each such pair, one of the two abstract mapping is removed, according to the test in the procedure remove-non-maximal. Since in each removal step, a single element is removed, there can be at most $x$ removal steps. Thus, the runtime for the removal of the non-maximal elements is in $O(x)$.

Consider a node $v$ in $Q$. MAMA computes $A_v$, according to three different cases. In all the three cases, searching $D$ for objects that have the same label as $v$ is in $O(d)$ runtime.

We now consider the three cases in MAMA for constructing $A_v$. If $v$ is a leaf (i.e., Case 1 of MAMA) then at most $d$ abstract mappings are added to $A_v$ and the evaluation is in $O(d)$.

Suppose that $v$ has a single child $v_1$ (i.e., Case 2 of MAMA). In the algorithm there is a loop over the objects of $D$ and a nested loop over the objects of $A_{v_1}$. In each inner loop, there is a test that checks if an object $o$ is contained in a set $S_1$. There are at most $d$ objects in $D$ with the same label as $v$. The set $A_{v_1}$ has at most $d$ elements. Thus, there are at most $d^2$ tests. Testing that $o$ is in $S_1$ is in $O(1)$, using the data structure we described above. Thus, the evaluation of this case is in $O(d^2)$ runtime.

Suppose that $v$ has $k > 1$ children. Then $A_v$ is computed in $k$ steps as described in Case 3 of MAMA. Each intermediate set $A'_{v_{i-1}}$ has at most $d$ objects because non-maximal elements are removed at the end of each step. Thus, in each step, two sets $A'_{v_{i-1}}$ and $A_{v_i}$, with at most $d$ elements, are combined. Combining the two sets is in $O(d^2)$ because of the same reasons that were given in the analysis of the runtime of Case 2. Since $k < q$, the evaluation of this case is in $O(qd^2)$ runtime.

All together, there are $q$ sets to compute. Each set is computed in $O(qd^2)$
runtime. Thus, the whole computation is in $O(q^2d^2)$ runtime.

\begin{algorithm} \textbf{SFTT} \\
So far, we described how to compute the maximal abstract mappings of variables in a tree query w.r.t. a tree database. Now, we will present Algorithm SFTT that actually computes semiflexible matchings of a plain tree query w.r.t. a tree database. We will also show that the algorithm has polynomial time complexity, in the size of the input and the output.

Before presenting the algorithm, we need to introduce a new definition. In Definition 3.26, matching properties were defined. We now define \textit{compliance} with matching properties.

\begin{definition} \textbf{(Compliance)} Suppose that $Q$ is a query and $D$ is a database. Consider a matching property $P = (v,o,V_P,S_P)$ w.r.t. $Q$ and $D$. Let $\mu$ be an assignment to $V_P$ w.r.t. $D$. We say that $\mu$ complies with $P$ if $\mu(u) \in S_P$ for each $u$ in $V_P$.

We now provide a description of SFTT. SFTT computes the semiflexible matchings using a top-down traversal over the query. Matchings are computed by a series of extension steps. In each step, the algorithm descends to a new variable and matchings that were computed in a previous step are extended w.r.t. the new variable. Maximal abstract mappings are being used as SF-properties and they make sure that the created matchings are semiflexible matchings.

Consider a tree query $Q$ and a tree database $D$. Algorithm SFTT computes a set $\mathcal{M}$ of the semiflexible matchings of $Q$ w.r.t. $D$ as follows.

1. In the first phase of the algorithm, the sets of maximal abstract mappings $A_v$ are computed by MAMA for all the variables $v$ of $Q$.

2. The variables of $Q$ are sorted topologically. Let $v_0,v_1,\ldots,v_n$ be the variables of $Q$ ordered according to the topological sort. Note that $v_0$ is the root of $Q$.

3. Iteratively, $n+1$ sets $\mathcal{M}^0,\ldots,\mathcal{M}^n$ are computed as described next.
4. If $A_{v_0}$ is not empty, then the set $\mathcal{M}^0$ consists of a single mapping $\mu_0 = \{(v_0, r_D)\}$, i.e., $\mu_0$ is the mapping of the query root $(v_0)$ to the database root. If $A_{v_0}$ is empty, then the algorithm stops and returns an empty set $\mathcal{M}$.

5. In step $i$, the set $\mathcal{M}^i$ is constructed from $\mathcal{M}^{i-1}$ by extending the matchings in $\mathcal{M}^{i-1}$ w.r.t. $v_i$, as follows.

   (a) Iteratively, each matching $\mu_{i-1}$ in $\mathcal{M}^{i-1}$ and each abstract mapping $(o, S)$ in $A_{v_i}$ are considered.

   (b) Let $P = (v_i, o, \{u \mid u \succ v_i\}, S)$ be a matching property that the abstract mapping $(o, S)$ defines. If $\mu_{i-1}$ complies with $P$ then $\mu_i = \mu_{i-1} \cup \{(v_i, o)\}$, i.e., $\mu_{i-1}$ is extended by mapping $v_i$ to $o$, and $\mu_i$ is added to $\mathcal{M}^i$.

6. Let $\mathcal{M}$ be the set $\mathcal{M}^n$. The set $\mathcal{M}$ is returned.

The following propositions prove the correctness of SFTT and analyze its time complexity.

**Proposition 3.42 (Correctness)** Suppose that $Q$ is a plain tree query and $D$ is a tree database. The set $\mathcal{M}$ that SFTT returns is the set of all the semiflexible matchings of $Q$ w.r.t. $D$.

**Proof.** There are two things to show. One thing to show is that all the matchings in $\mathcal{M}$ are semiflexible matchings of $Q$ w.r.t. $D$. The second thing to show is that all the semiflexible matchings of $Q$ w.r.t. $D$ are in $\mathcal{M}$.

First, we show that all the matchings in $\mathcal{M}$ are semiflexible matchings of $Q$ w.r.t. $D$. We show this by showing that all the matchings in $\mathcal{M}$ are connectivity-preserving mappings (Definition 3.10). This will prove what we want, according to Corollary 3.13 and the fact that in all the mappings in $\mathcal{M}$ the query root is mapped to the database root.

Consider a matching $\mu$ in $\mathcal{M}$. We show that the first condition of Definition 3.10 holds. Let $v_i$ be a variable of $Q$. Suppose that $Q$ has a simple path of the form $v_0 \ast lv_i$. Then, the object $\mu(v_i)$ is reachable from the root and has a single incoming label. This label must be $l$ because all the objects in $A_{v_i}$ have an incoming label
that is equal to the incoming label of \( v \). Thus, there is a path in \( D \) of the form 
\[
\tau_D \ast \mu(v_i).
\]

We show that the second condition of Definition 3.10 holds. Let \( v_i \) and \( v_j \) be two variables of \( Q \) such that there is a path in \( Q \) of the form \( v_h l_i v_i l_j v_j \). Consider step \( j \) in the algorithm, where the assignments of \( \mathcal{M}^{j-1} \) are extended w.r.t. \( v_j \).

Because \( \mu \) is created by the algorithm the following three assertions must be true. First, there is a matching \( \mu_{j-1} \) in \( \mathcal{M}^{j-1} \) such that \( \mu_{j-1}(v_i) = \mu(v_i) \), for each \( 1 \leq l < j \). Second, there is an abstract mapping \( (\mu(v_j), S) \) in \( A_{v_j} \). Third, \( \mu_{j-1} \) complies with the matching property \( P = (v_j, \mu(v_j), \{ u \mid u \succ v_j \}, S) \).

The variable \( v_i \) is above \( v_j \) in \( Q \). Hence, \( v_i \) is in the set \( \{ u \mid u \succ v_j \} \). From the compliance of \( \mu_{j-1} \) with \( P \), it follows that \( \mu_{j-1}(v_i) \) is in \( S \). We showed earlier that \( \mu_{j-1}(v_i) = \mu(v_i) \), for each \( 1 \leq l < j \). In particular, \( \mu_{j-1}(v_i) = \mu(v_i) \). So, \( \mu(v_i) \in S \).

The pair \( (\mu(v_j), S) \) is an abstract mapping of \( v_j \). According to the definition of abstract mappings (Definition 3.29), all the object in \( S \) are on a path with \( \mu(v_j) \). Since \( \mu(v_i) \) is in \( S \), \( \mu(v_i) \) and \( \mu(v_j) \) are on a path in \( D \). The label that enters \( \mu(v_i) \) and the label that enters \( \mu(v_j) \) are \( l_i \) and \( l_j \), respectively, because \( \mu(v_i) \) and \( \mu(v_j) \) are in abstract mappings of \( A_{v_i} \) and \( A_{v_j} \), respectively. Thus, \( D \) either has a path of the form \( \mu(v_i) \ast l_i \mu(v_j) \) or a path of the form \( \mu(v_j) \ast l_i \mu(v_i) \). This shows that the second condition of Definition 3.10 holds and \( \mu \) is a semiflexible matching.

Now, we show that all the semiflexible matchings of \( Q \) w.r.t. \( D \) are in \( M \). First, we show that the abstract mappings are not too restrictive.

Suppose that \( \mu \) is a semiflexible matching of \( Q \) w.r.t. \( D \). Let \( v \) be a variable of \( Q \). Then, according to Proposition 3.30, there is an abstract mapping of \( (\mu(v), S) \) is \( A_v \) such that if \( u \) is an ancestor of \( v \) then \( \mu(u) \in S \).

Let \( \mu_i \) \( (0 \leq i \leq n) \) denote a matching that is defined for the variables \( v_0, \ldots, v_i \), where \( \mu_i(v_j) = \mu(v_j) \) for each \( 0 \leq j \leq i \). We complete the proof by induction on \( i \).

The inductive hypothesis is that \( \mu_i \) is in \( \mathcal{M}^i \). For \( i = 0 \) the claim holds, since Proposition 3.30 implies that \( A_{v_0} \) is empty only if there are no semiflexible matchings of \( Q \) w.r.t. \( D \).

Suppose that the inductive hypothesis holds for \( i \). We want to show that it also holds for \( i + 1 \), i.e., \( \mu_{i+1} \) is in \( \mathcal{M}^{i+1} \).
Chapter 3 Flexible and Semiflexible Queries

According to the inductive hypothesis, \( \mu_i \) is in \( M^i \). As was shown above, there is an abstract mapping \( (\mu(v_{i+1}), S) \) in \( A_{v_{i+1}} \), such that \( \mu_i \) complies with the matching property \( P = (v_{i+1}, \mu(v_{i+1}), \{ u \mid u > v_{i+1} \}, S) \). Thus, \( \mu_i \) is extended by adding to it \( (v_{i+1}, \mu(v_{i+1})) \). Hence, \( \mu_{i+1} \) is in \( M^{i+1} \). This completes the proof of the induction and completes the proof of the proposition.

In the following, we analyze the time complexity of SFTT. First, we show that in the second phase of the algorithm there are no dangling matchings. (A matchings \( \mu_i \) in \( M^i \) is considered dangling if in the \( i+1 \)st step of the algorithm it is not extended with any assignment to \( v_i \).)

**Lemma 3.43 (No Dangling Matchings)** In SFTT, \( |M^i| \leq |M^{i+1}| \) for each \( 0 \leq i \leq n - 1 \).

**Proof.** We prove the lemma by showing that for every matching \( \mu_i \) in \( M^i \) there is a matching \( \mu_{i+1} \) that extends \( \mu_i \) and belongs to \( M^{i+1} \).

Suppose that \( \mu_i \) is in \( M^i \). Consider the variable \( v_{i+1} \). Let \( v_h \) be the parent of \( v_{i+1} \) in \( Q \).

In Step \( h \) of the algorithm, the mapping \( \mu_{h-1} \) was extended by assigning \( \mu_i(v_h) \) to \( v_h \). Thus, there is an abstract mapping \( (\mu_i(v_h), S_h) \) in \( A_{v_h} \), such that \( \mu(u) \in S_h \) holds for each variable \( u \) above \( v_h \).

As was claimed, \( (\mu_i(v_h), S_h) \) is an abstract mapping w.r.t. \( v_h \). In addition, \( v_{i+1} \) is a child of \( v_h \). Hence, according to the definition of the abstract mapping (Definition 3.29), there is an abstract mapping \( (o, S) \) of \( v_{i+1} \), such that \( S_h \subseteq S \) and \( \mu_i(v_h) \in S \). Thus, in \( A_{v_{i+1}} \) there is \( (o', S') \) that is either equal to or contains \( (o, S) \).

Let \( u \) be a variable above \( v_{i+1} \) in \( Q \). If \( u \) is \( v_h \) then \( \mu_i(u) \in S \). If \( u \) is above \( v_h \), then \( \mu_i(u) \in S_h \) and \( S_h \subseteq S \), so again \( \mu_i(u) \in S \). Consider the matching property \( P = (v_{i+1}, o', \{ u \mid u > v_{i+1} \}, S') \). Then, \( \mu_i \) complies with \( P \) and, hence, \( \mu_i \) is extended in the \( i+1 \)st step of the algorithm. That is, \( \mu_i \cup \{(v_{i+1}, o')\} \) is in \( M^{i+1} \).

This shows that there are no dangling matchings. Thus, the size of the sets \( M^i \) monotonically increases.
**Proposition 3.44 (Complexity)** Suppose that SFTT is computed w.r.t. a plain tree query $Q$ and a tree database $D$. SFTT has $O(|Q|^2 |D| (|D| + |M|))$ time complexity, where $|Q|$, $|D|$ and $|M|$ are the sizes of the query, the database and the result, respectively.

**Proof.** The first phase of the algorithm is in $O(|Q|^2 |D|)$, according to Proposition 3.40.

Consider extension step $i$ in the second phase of the algorithm. In this step, there is an iteration over the $|M^i|$ matchings of $M^i$ and the abstract mappings of $A_v$. By Lemma 3.43, $|M^i| \leq |M|$. By Proposition 3.40, the size of $A_v$ is at most $|D|$. Thus there are $O(|M||D|)$ iterations. It each iteration, an abstract mapping $(o, S)$ is tested w.r.t. the assignment to the $O(|Q|)$ variables that are above $v_i$ in $Q$. Recall that it is in $O(1)$ runtime to test if an object is in a set that is defined by a basic abstract set (assuming that a suitable data structure is initially constructed).

In the second phase of the algorithm there are $|Q|$ extension steps. According to the above arguments, each step is in $O(|M||D||Q|)$. Thus, the second phase of the algorithm is in $O(|Q|^2 |D||M|)$ runtime. The two phases together have $O(|Q|^2 |D| (|D| + |M|))$ time complexity. 

---

**Evaluating a DAG query over a Tree Database**

Consider a plain dag query $Q$ and a tree database $D$. In this section, we show how to compute the semiflexible matchings of $Q$ w.r.t. $D$ in polynomial time, under input-output complexity.

SFTT was presented in the previous section as an algorithm for computing matchings of a semiflexible tree query w.r.t. a tree database. SFTT remains sound and complete when the query is a dag. That is, SFTT can compute the semiflexible matchings of a dag query $Q$ w.r.t. a tree database $D$. However, for dag queries, SFTT does not have a polynomial-time input-output complexity. The cause of this is that in extension steps w.r.t. variables with more than one incoming edge, there could be dangling matchings.

Next, we describe how to modify SFTT so that the input-output time complexity
Chapter 3  Flexible and Semiflexible Queries

will be polynomial for plain dag queries. We start by presenting some required notations and definitions.

Definition 3.45 (Intersection Node, Origin, Middle node) A variable \( v \), in a dag query \( Q \), is an intersection node if there are two or more edges in \( Q \) that enter \( v \). A node \( u_s \) is a source of \( v \) if all the paths from the root to \( v \) go through \( u_s \). The origin of \( v \) is the lowest source of \( v \), i.e., a source of \( v \) that has no source of \( v \) below it. A node \( w \) is a middle node of \( v \) if it is between \( u_s \) and \( v \). That is, there is a path from \( u_s \) to \( w \) and a path from \( w \) to \( v \). The origin of \( v \) is also considered a middle node of \( v \).

It is easy to see that in a dag query, an intersection node has exactly one origin but a node can be the origin of several intersection nodes.

The following observation shows why evaluation of dag queries w.r.t. a tree database is not computationally hard. In a semiflexible matching of a dag w.r.t. a tree, for each intersection node \( v \), either all the nodes above \( v \) or all the nodes below \( v \) are mapped to a single database path. This is proved in the next proposition.

Proposition 3.46 (Mapping Intersection Nodes) Let \( Q \) be a dag query, \( D \) be a tree database and \( \mu \) be a semiflexible matching of \( Q \) w.r.t. \( D \). If \( v \) is an intersection node of \( Q \) then one of the next two statements must be true.

- For each pair of nodes \( u_1 \) and \( u_2 \) above \( v \), their images \( \mu(u_1) \) and \( \mu(u_2) \) are connected by a path.

- For each pair of nodes \( w_1 \) and \( w_2 \) below \( v \), their images \( \mu(w_1) \) and \( \mu(w_2) \) are connected by a path.

Proof. Consider a case where \( u_1 \) and \( u_2 \) are nodes above \( v \) such that \( \mu(u_1) \) and \( \mu(u_2) \) are not connected by a path. In addition, \( w_1 \) and \( w_2 \) are nodes below \( v \) such that \( \mu(w_1) \) and \( \mu(w_2) \) are not connected by a path. We show that the assumptions of this case lead to a contradiction.

In the query, there is a path from \( u_1 \) to \( v \) and a path from \( v \) to \( w_1 \). Thus, there is a path in \( Q \), from \( u_1 \) to \( w_1 \). Similarly, we can show that \( Q \) has a path from \( u_1 \) to \( w_2 \), a path from \( u_2 \) to \( w_1 \) and a path from \( u_2 \) to \( w_2 \).
Because there is a path in $Q$ from $u_1$ to $w_1$, either $D$ has a path from $\mu(u_1)$ to $\mu(w_1)$ or vice versa. Similarly, there must be a path from $\mu(u_1)$ to $\mu(w_2)$ or vice versa. We assumed that $\mu(w_1)$ and $\mu(w_2)$ are not connected by a path. Hence, $\mu(w_1)$ and $\mu(w_2)$ must be below $\mu(u_1)$, in $D$.

By replacing the roles of $u_1$ and $u_2$ with $w_1$ and $w_2$, we can show, in the same way, that $\mu(u_1)$ and $\mu(u_2)$ must be below $\mu(w_1)$, in $D$. However, $D$ is a tree, so it impossible that, simultaneously, $\mu(w_1)$ is below $\mu(u_1)$ and $\mu(u_1)$ is below $\mu(w_1)$. This contradiction completes the proof.\[\square\]

We will show how to construct mappings of $Q$ to $D$ according to the restriction that Proposition 3.46 imposes. Consider a dag query $Q$ and a tree database $D$. Let $v_0, \ldots, v_n$ be the variables of $Q$ and $A_{v_0}, \ldots, A_{v_n}$ be the sets of maximal abstract mappings that the procedure MAMA computes.

**Definition 3.47 (Restriction of Abstract Mappings)** Suppose that $(o_i, S_i)$ is an abstract mapping of an intersection node $v_i$. The restriction of $A_{v_0}, \ldots, A_{v_n}$ to $(o_i, S_i)$ and $v_i$ are the sets $A'_{v_0}, \ldots, A'_{v_n}$ such that the following holds.

1. The set $A'_{v_i}$ contains $(o_i, S_i)$ and no other element.

2. Consider a middle node $v_j$ of $v_i$. The set $A'_{v_i}$ satisfies the next three conditions.

   (a) $A'_{v_j} \subseteq A_{v_j}$.

   (b) For every $(o_j, S_j)$ in $A'_{v_j}$, it holds that $o_j \in S_i$.

   (c) Suppose that $v_j$ has children $v^1_j, \ldots, v^j_k$ in $Q$. Then there are abstract mappings $(o^1_i, S^1_i), \ldots, (o^j_k, S^j_k)$ of $v^1_j, \ldots, v^j_k$, respectively, such that, for each $1 \leq i \leq k$, $S_j \subseteq S^j_i$, $o_j \in S^j_i$ and $(o^j_i, S^j_i)$ is in $A'_{v_i}$.

3. For $v_j$ that is not a middle node of $v_i$, $A'_{v_j}$ is equal to $A_{v_j}$.

Consider the case where in Definition 3.47, $S_i$ has the form of $\{ x \mid x \leq o' \vee x \geq o' \}$. Then, the sets of abstract mappings in the restriction correspond to mappings in which the middle nodes of $v$ are mapped to database objects on the path from the database root to $o'$. Suppose that $S_i$ has the form of $\{ x \mid x \geq o' \}$. Then, the sets
of abstract mappings in the restriction correspond to mappings in which the nodes below $v$ are mapped to objects of $D$ that are above $v'$.

Consider a plain dag query $Q$, a tree database $D$ and a semiflexible matching $\mu$ of $Q$ w.r.t. $D$. Let $v_0, \ldots, v_n$ be the variables of $Q$ and $v_i$ be an intersection node. In addition, let $v_{i_1}, \ldots, v_{i_m}$ be the middenodes of $v_i$.

**Proposition 3.48 (Completeness)** Suppose that $(\mu(v_i), S_i)$ is an element of $A_{v_i}$. Let $A'_{v_0}, \ldots, A'_{v_n}$ be the restriction of $A_{v_0}, \ldots, A_{v_n}$ to $(\mu(v_i), S_i)$. Then, for each $1 \leq i \leq n$, there is $(\mu(v_i), S_i)$ in $A'_v$ such that for every variable $u$ above $v_i$, $\mu(u)$ is in $S_i$.

**Proof.** If $v_i$ is not an intersection node of $v_i$, then $A'_{v_i}$ is equal to $A_{v_i}$. In this case, the claim follows from Proposition 3.30.

If $v_i$ is an intersection node of $v_i$, then $\mu(v_i)$ satisfies $S_i$. Thus, a simple induction shows that each maximal abstract mappings $(\mu(v_i), S_i)$ of $A_{v_i}$ is also in $A'_{v_i}$.

**Definition 3.49 (Dangling Abstract Sets)** Suppose that $(o_i, S_i)$ is an abstract mapping of an intersection node $v_i$. We say that $(o_i, S_i)$ is dangling if in the restriction of $A_{v_0}, \ldots, A_{v_n}$ to $(o_i, S_i)$ and $v_i$, one (or more) of the sets is empty.

In order to avoid dangling assignments, the extensions of assignments in the algorithm should be computed w.r.t. abstract mappings that are not dangling, as will be explained when we will describe the algorithm.

Consider a dag query $Q$ and a tree database $D$. Let $v_0, \ldots, v_n$ be the variables of $Q$ ordered topologically. Let $v_i$ be an intersection node and $v_{i_1}, \ldots, v_{i_m}$ be the middenodes of $v_i$. Let $A_{v_0}, \ldots, A_{v_n}$ be the sets of maximal abstract mappings of $v_0, \ldots, v_n$, respectively. Finally, let $(o_i, S_i)$ be an abstract mapping in $A_{v_i}$ and let $A'_{v_0}, \ldots, A'_{v_n}$ be the restriction of $A_{v_0}, \ldots, A_{v_n}$ to $(o_i, S_i)$ and $v_i$.

**Definition 3.50 (Compliance)** Suppose that $\mu$ is a mapping of $v_{i_1}, \ldots, v_{i_l}$ ($l \leq m$) to $D$. We say that $\mu(v_{i_1}), \ldots, \mu(v_{i_l})$ comply with $(o_i, S_i)$ and $v_i$ if there are $(\mu(v_{i_1}), S_i), \ldots, (\mu(v_{i_l}), S_i)$ in $A'_{v_0}, \ldots, A'_{v_n}$, respectively.
Chapter 3  Flexible and Semiflexible Queries

Now, we show how to modify algorithm SFTT for dealing with dag queries. We call the modified algorithm SFDT. (Notice that the difference in the names of the algorithms is in the third letter, where D replaces T.)

Consider a plain dag query. Let $v_0, \ldots, v_n$ be the variables of $Q$. Let $D$ be a tree database. SFDT performs the following steps.

1. As a preliminary step, for each intersection node $v_i$, it is verified that $v_i$ does not have two incoming edges with different labels. In addition, the origin of $v_i$, denoted $v_o$, is discovered and the set $M_{v_i}$ of the middle nodes of $v_i$ is computed.

2. A topological order $v_0, \ldots, v_n$ of the nodes of $Q$ is computed.

3. The sets $A_{v_0}, \ldots, A_{v_n}$ are computed using MAMA.

4. Consider an intersection node $v_i$ and an abstract mapping $(o_i, S_i)$ of $v_i$, in $A_{v_i}$. The restriction of $A_{v_0}, \ldots, A_{v_n}$ to $(o_i, S_i)$ and $v_i$ is computed. The computation is similar to the computation of the sets $A_{v_0}, \ldots, A_{v_n}$, except that for the middle nodes of $v_i$, abstract mappings are computed merely w.r.t. nodes that are in $S_i$. If, at the end of the computation, $(o_i, S_i)$ is dangling then it is discarded from $A_{v_i}$.

5. The sets $M^0, \ldots, M^n$ are computed as in SFTT with the following difference. Consider an assignment $\mu_{j-1}$ in $M^{j-1}$ and an abstract mapping $(o, S)$ in $A_{v_j}$.

(a) Suppose that $v_j$ is a middle node of an intersection node $v_i$. In addition, let $v_{i_1}, \ldots, v_{i_k}$ be the middle nodes of $v_i$ that appear before $v_j$ in a topological order of Step 2. Then, $\mu_{j-1}$ is extended by mapping $v_j$ to $o$ only if $\mu_{j-1}$ complies with the matching property that $(o, S)$ defines and, in addition, $\mu(v_{i_1}), \ldots, \mu(v_{i_k})$ and $\mu(v_j)$ comply with $(o, S)$ and $v_i$.

(b) If $v_j$ is not a middle node of any node, then the extension is computed as in SFTT.

**Proposition 3.51 (Correctness)** Suppose that $Q$ is a plain tree query and $D$ is a tree database. Algorithm SFDT returns the set of semiflexible matchings of $Q$ w.r.t. $D$. 

Proof. First, we claim that all the matchings returned by the algorithm are semiflexible matchings. This is proved in a similar way to the proof of Proposition 3.42. Essentially, the proof shows that in each extension step the assignments comply with the SF-Property that the abstract matchings define. Thus, the created assignments are semiflexible matchings.

Secondly, we claim that all the semiflexible matchings of $Q$ w.r.t. $D$ are computed by the algorithm. The difference between $SFDT$ and $SFTT$ is in adding a condition of compliance with the abstract matchings of the intersection nodes. Proposition 3.48 shows that this condition does not discard semiflexible matchings. The rest of the proof is similar to the proof of Proposition 3.42. □

Before we can show that $SFDT$ has polynomial input-output time complexity, we need to show that there are no dangling matchings, and thus, each intermediate set $\mathcal{M}^i$ is not larger than the output of the algorithm.

**Proposition 3.52 (Monotonic Increase)** In Algorithm $SFDT$, $|\mathcal{M}^i| \leq |\mathcal{M}^{i+1}|$, for each $0 \leq i \leq n$.

**Proof.** We show that the algorithm does not produce dangling matchings. Consider a matching $\mu_i$ in $\mathcal{M}^i$. We need to show that there is a matching $\mu_{i+1}$ in $\mathcal{M}^{i+1}$ such that $\mu_i(v_h) = \mu_{i+1}(v_h)$, for each $1 \leq h \leq n$.

If $v_{i+1}$ is not an intersection node then the proof is similar to the proof of 3.43, with the next addition. The condition of compliance with the abstract matchings of the intersection nodes does not prevent extending $\mu_i$ with an assignment of $v_{i+1}$ to $\mu_{i+1}(v_{i+1})$ due to Proposition 3.48.

Suppose that $v_{i+1}$ is an intersection node. Then $\mu_i$ assigns the middlenodes of $v_{i+1}$ to objects that comply with some abstract mapping $(o, S)$ of $A_{v_{i+1}}$. In this case, $\mu_i$ can be extended by assigning $o$ to $v_{i+1}$. □

Finally, we show that the runtime of $SFDT$ is polynomial in the size of the input and the output.

**Theorem 3.53 (Complexity)** Let $Q$ be a plain dag query and let $D$ be a tree database. Algorithm $SFDT$ computes the set of semiflexible matchings $\mathcal{M}_f(Q, D)$
in polynomial time, in the size of the input and the output.

Proof. Let \( q, d \) and \( m \) be the sizes of the query, the database and the result, respectively.

Computing the sets \( A_{v_0}, \ldots, A_{v_n} \) is in \( O(q^2d^2) \), according to Proposition 3.40. Computing the restrictions of \( A_{v_0}, \ldots, A_{v_n} \) to abstract mappings of intersection nodes is in \( O(q^3d^3) \). This is because there are at most \( q \) intersection nodes. There are at most \( d \) abstract mappings in \( A_{v_i} \), for each intersection node \( v_i \). Thus, there are \( O(qd) \) computations of restriction and each computation is in \( O(q^2d^2) \), i.e., has complexity that is similar to the complexity of computing \( A_{v_0}, \ldots, A_{v_n} \).

There are \( O(q) \) extension steps in the algorithm. In each extension step, at most \( m \) matchings are extended, according to Proposition 3.52. Each extension step requires checking two things. First, compliance with the matching property. This is tested in \( O(q) \), since there are at most \( q \) variables above each variable of the query. The second thing to check is compliance with an abstract mapping of an intersection node. This is tested in \( O(qd) \) because there are \( d \) possible abstract mapping to comply with and there are at most \( q \) middlenodes for each intersection variable.

All together, \( SFDT \) has \( O(q^3d^3 + dm(q + d)) \) time complexity. Therefore, computing the semiflexible matchings of \( Q \) w.r.t. \( D \) has polynomial time complexity, in the size of the input and the output. \( \square \)

### 3.2.3 Cyclic Databases and Cyclic Queries

For the case of a cyclic database, deciding if there exists a semiflexible matching of a given query w.r.t. the database is NP-Complete. NP-Hardness follows from Theorem 3.23. For acyclic queries, Membership in NP follows from Theorem 3.19. For cyclic queries, since the result of evaluating a cyclic query over a dag database is always empty, NP-Hardness does not follow from Theorem 3.23. However, a reduction similar to that of Theorem 3.23 shows that nonemptiness of a cyclic query over a cyclic database is NP-Hard.

The complexity results for the various cases are shown in Table 3.2. Note that the
Table 3.2: The complexity of checking nonemptiness under the semiflexible semantics.

cases for which it is said that testing emptiness is in polynomial time (in the size of the query and the data), their query-evaluation has a polynomial-time input-output complexity. When testing emptiness is not polynomial, obviously, query-evaluation cannot have a polynomial-time input-output complexity either.

3.3 Evaluating Flexible Queries

In this section, we discuss query evaluation under the flexible semantics. First, we will show that query evaluation under the flexible semantics can be reduced to query evaluation under the rigid semantics.

Definition 3.54 (Reachability Graph) Let $D = (O, E, r_D, \alpha)$ be a database over the set of nodes $O$. The reachability graph of $D$ is a rooted labeled directed graph, denoted $RG(D) = (O, E_R, r_D, \alpha)$, that is obtained from $D$ by adding edges as follows. If object $o'$ of $D$ has an incoming edge labeled with $l$ and there is either a path from $o'$ to another object $o$ or vice-versa, then $E_R$ has an edge labeled with $l$ from $o$ to $o'$.

If a database $D$ contains $n$ nodes and the ingoing degree of $D$ is $k$ (the ingoing degree of $D$ is the maximal number of edges that enter a node in $D$) then $RG(D)$ has at most $n^2 \cdot 2k$ edges. The reason for this is that between each pair of nodes in $RG(D)$ there can be at most $2k$ edges with different labels. Hence, the size of $RG(D)$ is polynomial in the size of $D$. Note that the reachability graph can be stored in a data structure whose size is $n^2 + n \cdot k$. In the data structure, for each pair of nodes there should be a bit that indicates whether there is a path between these nodes. In addition, each node should have a list with the labels that enter it.
Theorem 3.55 (Reduction to the Rigid Semantics) Consider a query $Q$ and a database $D$. Let $RG(D)$ be the reachability graph of $D$. The following two sets are equal.

- The set of flexible matchings of $Q$ w.r.t. $D$.
- The set of rigid matchings of $Q$ w.r.t. $RG(D)$.

Proof. First, we show that every flexible matching of $Q$ w.r.t. $D$ is also a rigid matching of $Q$ w.r.t. $RG(D)$. According to Definition 3.5, if $\mu$ is a flexible matching of $Q$ w.r.t. $D$, then it satisfies the rc, all the wcc’s and all the qcc’s of $Q$. We need to show that $\mu$ satisfies all the ec’s of $Q$ w.r.t. $RG(D)$.

Let $ulv$ be an ec of $Q$. The matching $\mu$ satisfies the wcc $lv$ w.r.t. $D$. Thus, the object $\mu(v)$ has an incoming label $l$ in $D$. In addition, $\mu$ satisfies the qcc $ulv$ w.r.t. $D$. So, either there is a path in $D$ from $\mu(v)$ to $\mu(u)$ or vice-versa. According to Definition 3.54, $RG(D)$ has an edge $\mu(u)\mid\mu(v)$, and hence the ec $ulv$ is satisfied w.r.t. $RG(D)$. From the above, it follows that $\mu$ is a rigid matching of $Q$ w.r.t. $RG(D)$. This is because, w.r.t. $RG(D)$, $\mu$ satisfies the rc and all the ec’s of $Q$.

For the other direction, we show that every rigid matching of $Q$ w.r.t. $RG(D)$ is also a flexible matching of $Q$ w.r.t. $D$. Let $\mu$ be a rigid matching of $Q$ w.r.t. $RG(D)$. The matching $\mu$ satisfies the rc and all the ec’s of $Q$ w.r.t. $RG(D)$. We need to show that $\mu$ satisfies all the wcc’s and all the qcc’s of $Q$ w.r.t. $D$.

Let $lv$ be a wcc in $Q$. The wcc $lv$ is part of an ec $ulv$ for some node $u$. The matching $\mu$ satisfies the ec $ulv$ w.r.t. $RG(D)$. Thus, there is an edge $\mu(u)\mid\mu(v)$ in $RG(D)$. It follows that $\mu(v)$ has an incoming edge labeled with $l$. So, $\mu$ satisfies the wcc $lv$ w.r.t. $D$. This shows that $\mu$ satisfies all the wcc’s of $Q$.

We now show that, w.r.t. $D$, $\mu$ satisfies all the qcc’s of $Q$. Let $ulv$ be a qcc of $Q$. The matching $\mu$ satisfies the ec $ulv$ of $Q$ w.r.t. $RG(D)$. So, $RG(D)$ contains the edge $\mu(u)\mid\mu(v)$. This means that either there is a path in $D$ from $\mu(u)$ to $\mu(v)$ or vice-versa. Thus, $\mu$ satisfies the qcc $ulv$ w.r.t. $D$. This shows that $\mu$ satisfies all the qcc’s of $Q$. Since, w.r.t. $D$, $\mu$ satisfies the rc, all wcc’s and all the qcc’s of $Q$, it is a flexible matching of $Q$ w.r.t. $D$. 

\qed
Chapter 3  Flexible and Semiflexible Queries

Evaluating all the rigid matching of $Q$ w.r.t. $RG(D)$ can be done as follows. For each edge $e = uv$ of $Q$, we create a binary edge relation $r_e$ that has the attributes $u$ and $v$, and the following set of tuples: $\{(o, o') \mid RG(D)$ has an edge $o'd\}$. In other words, $r_e$ contains all edges of $RG(D)$ that have the same label as $e$. The join of all the edge relations yields all the rigid matchings of $RG(D)$. When the join is acyclic, Yannakakis’s algorithm [65] can be applied and, hence, we get the following corollary.

**Corollary 3.56 (Tree Evaluation is in Polynomial Time)** When the query is a tree and the database is any graph, query evaluation under the flexible semantics has a polynomial-time input-output complexity.

Let $\mathcal{M}_f(Q, D)$ denote the set of flexible matchings of a query $Q$ w.r.t. a database $D$. The next lemma shows that if $Q$ is not a tree then it is NP-hard to decide whether $\mathcal{M}_f(Q, D)$ is not empty. We use a reduction of 3SAT.

**Lemma 3.57 (Reduction of 3SAT)** For a given 3CNF formula $\varphi$ over a set of propositional letters $P$, one can construct in polynomial time a dag query $Q$ and a database $D$ such that the following are equivalent.

1. There is a flexible matching of $Q$ w.r.t. $D$;
2. There is an assignment for the propositional letters in $P$ that satisfies $\varphi$.

The proof of Lemma 3.57 is given in Appendix C.

The next theorem shows that if $Q$ is not a tree, then a query-evaluation algorithm with a polynomial-time input-output complexity is not likely to exist.

**Theorem 3.58 (DAG Evaluation is NP-Complete)** Given a dag query $Q$ and a database $D$, deciding whether the flexible semantics yields a nonempty result is NP-complete.

**Proof.** NP-hardness follows from Lemma 3.57. Membership in NP follows because a flexible matching can be guessed and verified in polynomial time by checking the satisfaction of the constraints with respect to the reachability graph of $D$. Recall that the size of the reachability graph of $D$ is polynomial in the size of $D$. \(\square\)
3.4 Query Containment and Query Equivalence

In the case of relational conjunctive queries, the final step of evaluation is a projection of the matchings onto the distinguished variables. Consequently, containment is defined in terms of those projections. In the case of queries over semistructured data, the final step is a construction of the result from the matchings. A discussion of this step, however, is beyond the scope of this work. Thus, in this work, the semantics of queries is defined in terms of matchings over all the variables, and containment (equivalence) is defined as containment (equality) of the corresponding sets of matchings.

Definition 3.59 (Query Containment, Query Equivalence) Let $Q_1$ and $Q_2$ be two queries over the same set of node variables and with the same root. We say that $Q_1$ is contained in $Q_2$ under the semantics $s$, denoted $Q_1 \subseteq_s Q_2$, if for all database $D$, $M_s(Q_1, D) \subseteq M_s(Q_2, D)$. The queries $Q_1$ and $Q_2$ are equivalent if for all database $D$, $M_s(Q_1, D) = M_s(Q_2, D)$.

Deciding equivalence and containment of queries is useful for optimization techniques. The next theorem provides a characterization of containment for semiflexible queries.

Theorem 3.60 (Semiflexible Containment) Let $Q_1$ and $Q_2$ be queries over the same set of variables $V$. $Q_1 \subseteq_{sf} Q_2$ if and only if the identity mapping $\nu$ over $V$ is a semiflexible matching of $Q_2$ w.r.t. $Q_1$.

Proof. Consider the case where the identity mapping $\nu$ is a semiflexible matching of $Q_2$ w.r.t. $Q_1$. Let $D$ be a database, and let $\mu \in M_{sf}(Q_1, D)$ be a semiflexible matching of $Q_1$ w.r.t. $D$. We want to show that $\mu$ is a semiflexible matching of $Q_2$ w.r.t. $D$.

Obviously, $\mu$ maps the root of the $Q_2$ to the root of $D$, since the root of $Q_2$ and the root of $Q_1$ are the same node.

Let $\pi_2 = v_0 l_1 v_1 \ldots l_n v_n$ be a path in $Q_2$. Since $\pi_2$ satisfies the SF-Condition w.r.t. $\nu$, there is a permutation $\sigma$ of $0, 1, \ldots, n$ such that $\sigma(0) = 0$ and $Q_1$ has
a path $\pi_1$ of the form $\nu(v_{\sigma(0)})\#_{l_{\sigma(1)}}\nu(v_{\sigma(1)})\#_{l_{\sigma(1)}}\nu(v_{\sigma(n)})$. But $\nu$ is the identity mapping so the form of $\pi_1$ is actually $v_{\sigma(0)}\#_{l_{\sigma(1)}}v_{\sigma(1)}\#_{l_{\sigma(n)}}v_{\sigma(n)}$. The path $\pi_1$ satisfies the SF-Condition w.r.t. $\mu$, because $\mu$ is a semiflexible matching of $Q_1$ w.r.t. $D$. So, there is a permutation $\sigma'$ such that $D$ includes a path with the form $\mu(v_{\sigma'(0)})\#_{l_{\sigma'(1)}}\mu(v_{\sigma'(1)})\#_{l_{\sigma'(n)}}\mu(v_{\sigma'(n)})$. This path shows that $\pi_2$ satisfies the SF-Condition w.r.t. $\mu$.

Given a strongly connected component $C$ in $Q_2$, $C$ is also a strongly connected component in $Q_1$ since $\nu$ is a semiflexible matching. Furthermore, the image of $C$ is a strongly connected component in $D$ since $\mu$ maps strongly connected components of $Q_1$ to strongly connected components of $D$. We conclude that $\mu$ is a semiflexible matching of $Q_2$ w.r.t. $D$.

For the other direction, assume that $Q_1 \subseteq_{sf} Q_2$. We create a database from $Q_1$ by adding to $Q_1$ some arbitrary function $\alpha$ that maps the atomic nodes of $Q_1$ to values (see Definition 2.1 and Definition 2.3). The identity mapping $\nu$ over $V$ is a semiflexible matching of $Q_1$ w.r.t. $Q_1$. Thus, $\nu$ is also a semiflexible mapping of $Q_2$ w.r.t. $Q_1$.  

\begin{corollary}[Polynomial Time Complexity] Deciding if $Q_1 \subseteq_{sf} Q_2$ is in polynomial time.
\end{corollary}

\begin{proof}
According to Theorem 3.19, the time complexity for verifying that a matching is a semiflexible matching is polynomial.
\end{proof}

A characterization of containment for flexible queries is given in the next theorem.

\begin{theorem}[Flexible Containment]
Let $Q_1$ and $Q_2$ be queries over the same set of variables. $Q_1 \subseteq_f Q_2$ if and only if the following two conditions hold.

1. For each weak $lv$ in $Q_2$, there is a weak $lv$ in $Q_1$.

2. For each $pure$ $uv$ in $Q_2$, where $u \neq r_{Q_2}$ and $v \neq r_{Q_2}$ ($r_{Q_2}$ is the root of $Q_2$),
   the query $Q_1$ contains either the $pure$ $ul'v$ or the $pure$ $vl'u$ for some label $l'$.
\end{theorem}
Chapter 3 Flexible and Semiflexible Queries

Proof. Assume that Condition 1 and Condition 2 hold for \( Q_1 \) and \( Q_2 \). Let \( D \) be a database and let \( \mu \) be a flexible matching of \( Q_1 \) w.r.t. \( D \). We show that \( \mu \) is a flexible matching of \( Q_2 \).

The root of \( Q_1 \) and the root of \( Q_2 \) are the same node. Thus, satisfaction of the rc of \( Q_1 \) is the same as satisfaction of the rc of \( Q_2 \). Let \( lv \) be a wec in \( Q_2 \). According to Condition 1, \( lv \) is also a wec in \( Q_1 \). Thus, \( \mu \) satisfies \( lv \). Let \( ulv \) be a qec of \( Q_2 \) such that both \( u \) and \( v \) are not \( \tau_{Q_2} \). By Condition 2, \( Q_1 \) contains either a qec \( ulv \) or a qec \( ulv \). In both cases, from satisfaction of the qec’s of \( Q_1 \) it follows that there is a path in \( D \) from \( \mu(v) \) to \( \mu(u) \) or vice-versa. This shows satisfaction of the qec \( ulv \) of \( Q_2 \). For a qec \( \tau_{Q_2}lv \) of \( Q_2 \), \( \mu(\tau_{Q_2}) \) is the root of \( D \) and obviously, there is a path from the root of \( D \) to any node of \( D \), including \( \mu(v) \). Thus, \( \mu \) satisfies this qec. For a qec \( ul\tau_{Q_2} \), the object \( \mu(\tau_{Q_2}) \) is the root of \( D \) and obviously, there is a path from the root of \( D \) to any node of \( D \), including \( \mu(u) \). Thus, \( \mu \) satisfies this qec. To conclude, \( \mu \) in a flexible matching of \( Q_2 \) since it satisfies the rc, the qec’s and the wec’s of \( Q_2 \).

For the other direction, we assume that \( Q_1 \subseteq f Q_2 \) and we show that the above two conditions hold.

To show that the first condition holds, we create a database from \( Q_1 \) by adding to \( Q_1 \) some arbitrary function \( \alpha \) that maps the atomic nodes of \( Q_1 \) to values (see Definition 2.1 and Definition 2.3). Obviously, the identity mapping \( \nu \) is a flexible matching of \( Q_1 \) w.r.t. \( Q_1 \). Since \( Q_1 \subseteq f Q_2, \nu \) is a flexible matching of \( Q_2 \) w.r.t. \( Q_1 \). Let \( lv \) be a wec in \( Q_2 \). Since \( \nu \) satisfies \( lv \) w.r.t. \( Q_1 \), there must be an edge in \( Q_1 \) that enters \( v \) and its label is \( l \). This means that there is a wec \( lv \) in \( Q_1 \) and Condition 1 holds.

To show that the second condition holds, consider a qec \( ulv \) in \( Q_2 \), where \( u \neq \tau_{Q_2} \) and \( v \neq \tau_{Q_2} \). We create a database \( D \) whose nodes are the variables of \( Q_1 \) and its root is the root of \( Q_1 \). For each label \( l \) in \( Q_1 \) and every two nodes \( w_1, w_2 \), such that \( w_1 \neq u \) and \( w_1 \neq v \) (for \( i = 1, 2 \), \( D \) includes an edge \( w_1lw_2 \). In addition, for each label \( l \) and node \( w \) such that \( w \neq u \) and \( w \neq v \), \( D \) consists of the edges \( vlu \) and \( wlu \).
Chapter 3  Flexible and Semiflexible Queries  66

The database \( D \) has the following three properties. First, there is a path between
each pair of nodes, except for the pair \( u \) and \( v \). Second, from each node there is a
path to \( u \) (\( v \)) but there is no path from \( u \) (\( v \)) to any node of \( D \). Third, for each
label \( l \) and each variable \( w \) of \( Q_1 \), \( D \) has an edge, with the label \( l \), that enters \( w \).
Note that in the construction of \( D \) we assumed that neither \( v \) nor \( u \) is the root of
\( Q_1 \). Thus, it is not necessary to have paths from from either \( u \) or \( v \) to all the other
nodes.

Since \( D \) has no path from \( u \) to \( v \) or vice-versa, there is no flexible matching of
\( Q_2 \) w.r.t. \( D \). We assumed that \( Q_1 \subseteq_f Q_2 \), so there is no flexible matching of \( Q_1 
\) w.r.t. \( D \). According to the construction of \( D \) the \( \text{rc} \) and all the \( \text{wec}'s \) of \( Q_1 \)
are satisfied by the identity mapping. Furthermore, every \( \text{qec} \), except for a \( \text{qec} \) \( ul'v \) or
a \( \text{qec} \) \( vl'u \), is satisfied by the identity mapping. Since the identity mapping is not
a flexible matching of \( Q_1 \) w.r.t. \( D \), \( Q_1 \) must include either the \( \text{qec} \) \( ul'v \) or the \( \text{qec} 
\) \( vl'u \), for some label \( l' \).

To decide containment of \( Q_1 \) in \( Q_2 \), under the flexible semantics, we just need
to check that every \( \text{wec} \) (\( \text{qec} \)) in \( Q_1 \) has a suitable \( \text{wec} \) (\( \text{qec} \)) in \( Q_2 \). Thus, we get
the following result.

Corollary 3.63 (Polynomial Time Complexity) Deciding whether \( Q_1 \subseteq_f Q_2 
\) is in \( O(|Q_1| \cdot |Q_2|) \) time.

If we sort the \( \text{rc}'s \) and \( \text{wec}'s \) of \( Q_2 \), deciding containment can be done in \( O(|Q_2| \cdot 
\log |Q_2| + |Q_1| \cdot \log |Q_2|) \).

3.5 Database Equivalence

In this section we introduce the novel notion of database equivalence and we charac-
terize database equivalence under the different semantics.

Definition 3.64 (Database Equivalence) Given two databases \( D \) and \( D' \) over
the same set of objects \( O \), we say that \( D \) and \( D' \) are equivalent under the semantics
\( s \) if for every query \( Q \), the set of \( s \)-matchings of \( Q \) w.r.t. \( D \) is equal to the set of
\( s \)-matchings of \( Q \) w.r.t. \( D' \).
Chapter 3  Flexible and Semiflexible Queries

Under the classical (i.e., rigid) semantics, two databases are equivalent if and only if they are isomorphic (i.e., have the same root and the same set of labeled edges). In the case of the semiflexible and flexible semantics, however, two databases can be equivalent even if they are not isomorphic.

One reason for investigating database equivalence is that in some cases it is more efficient to evaluate queries over databases that have a certain form than over databases that have a different form. For example, we showed that it is more efficient to evaluate queries over a tree database than over a dag database. Thus, it is important to be able to characterize equivalence of databases, and to be able to transform a database of a given form (e.g., dag) to an equivalent database that has a different form (e.g., tree).

Let $D$ and $D'$ be two databases that have the same set of objects and the same root. We say that a path $\phi = o_0 l_1 o_1 l_2 o_2 \cdots l_n o_n$ of $D$, where $o_0$ is the root, is included in a path $\phi'$ of $D'$ if there is a permutation $\sigma$ of $1, \ldots, n$, such that $\phi'$ has the form $o_0 l_{\sigma(1)} o_{\sigma(1)} l_{\sigma(2)} o_{\sigma(2)} \cdots l_{\sigma(n)} o_{\sigma(n)}$.

We say that there is a semiflexible path inclusion of $D$ in $D'$ if for every path $\phi$ in $D$ that starts at the root, there is a path $\phi'$ in $D'$, such that $\phi'$ includes $\phi$.

**Theorem 3.65 (Semiflexible Equivalence)** Consider two databases $D$ and $D'$ over the same set of objects and with the same root. $D$ and $D'$ are equivalent under the semiflexible semantics if and only if the following conditions hold.

1. There is a semiflexible path inclusion of $D$ in $D'$ and vice-versa.

2. Each strongly connected component in $D$ is a strongly connected component in $D'$ and vice-versa.

**Proof.** Suppose that the conditions of the theorem hold. We need to show that $D$ and $D'$ are equivalent under the semiflexible semantics. Let $Q$ be a query and $\mu \in \mathcal{M}_{sf}(Q, D)$ be a semiflexible matchings of $Q$ w.r.t. $D$. Since $D$ and $D'$ have the same set of objects, $\mu$ is an assignment of $Q$ w.r.t. $D'$. We show that $\mu$ is a semiflexible matching of $Q$ w.r.t. $D'$ by showing that it satisfies the conditions of Definition 3.4.
Chapter 3  Flexible and Semiflexible Queries  68

Since $D$ and $D'$ have the same root, $\mu$ maps the root of $Q$ to the root of $D'$, and thus, the rc of $Q$ is satisfied.

Given a path $\pi = v_0v_1\cdots v_nv_n$ in $Q$, $\mu$ satisfies the SF-Condition w.r.t. $D$ and hence there is a path $\phi$ in $D$ of the form $\mu(v_0)\circ l_{\sigma(1)}\mu(v_{\sigma(1)})\cdots l_{\sigma(n)}\mu(v_{\sigma(n)})$.
According to Condition 1, there is a semiflexible path inclusion of $D$ in $D'$ and, thus, $D'$ has a path $\mu(v_{\sigma'(0)})\circ l_{\sigma'(1)}\mu(v_{\sigma'(1)})\cdots l_{\sigma'(n)}\mu(v_{\sigma'(n)})$, for some permutation $\sigma'$ of $1,\ldots,n$. This shows that $\pi$ satisfies the SF-Condition w.r.t. $D'$.

If $C$ is a strongly connected component in $Q$ then it is mapped by $\mu$ to a strongly connected component of $D$. This strongly connected component of $D$ is also a strongly connected component of $D'$, according to Condition 2.

We have shown that $\mu$ satisfies the conditions of Definition 3.4 and hence it is a semiflexible matching of $Q$ w.r.t. $D'$. Exchanging $D$ with $D'$ shows that if $\mu$ is a semiflexible matching of $Q$ w.r.t. $D'$ then it is also a semiflexible matching of $Q$ w.r.t. $D$. To conclude, $D$ and $D'$ are equivalent under the semiflexible semantics.

For the other direction, suppose that $D$ and $D'$ are equivalent. Let $Q_D$ be a query that is created from $D$ by considering the objects of $D$ as variables and removing the atomic values. Note that the set of labeled edges of $D$ is equal to the set of labeled edges of $Q_D$.

The identity mapping $\mu_D$, from the variables of $Q_D$ to the objects of $D$, is a semiflexible matching of $Q_D$ w.r.t. $D$. Since $D$ and $D'$ are equivalent, $\mu_D$ is also a semiflexible matching of $Q_D$ w.r.t. $D'$.

Suppose the $\phi$ is a path in $D$. Then, $\phi$ is also a path in $Q_D$. The matching $\mu_D$ satisfies the SF-Condition w.r.t. $\phi$ and $D'$. Hence, there is a path $\phi'$ in $D'$ such that $\phi'$ is a permutation of $\phi$. That is, there is a semiflexible path inclusion of $D$ in $D'$.

Because $\mu_D$ is a semiflexible matching, it satisfies Condition 3 of Definition 3.4. Thus, every strongly connected of $Q_D$ (and hence of $D$) is a strongly connected component of $D'$.

By replacing the roles of $D$ and $D'$ we can show, in the same way, that there is a semiflexible path inclusion of $D'$ in $D$ and that every strongly connected component in $D'$ is a strongly connected component of $D$. This shows that the two conditions of the theorem hold. $\square$
Chapter 3  Flexible and Semiflexible Queries

Conditions (1) and (2) of the above theorem are essentially equivalent to the condition that the identity mapping (of the objects of $D$ to the objects of $D'$) is a semiflexible matching. Therefore, we get the following corollary.

**Corollary 3.66 (Time Complexity)** Under the semiflexible semantics, deciding database equivalence is in polynomial time.

For the flexible semantics, we have the following theorem.

**Theorem 3.67 (Flexible Equivalence)** Consider two databases $D$ and $D'$ over the same set of objects $O$ and with the same root. The databases $D$ and $D'$ are equivalent under the flexible semantics if and only if their reachability graphs are isomorphic, i.e., have the same set of labeled edges.

*Proof.* Consider the case where the reachability graphs of $D$ and $D'$ are isomorphic. Let $Q$ be a query. Because of the isomorphism, $\mathcal{M}_r(Q, RG(D)) = \mathcal{M}_r(Q, RG(D'))$. Theorem 3.55 shows that $\mathcal{M}_r(Q, RG(D)) = \mathcal{M}_f(Q, D)$ and $\mathcal{M}_r(Q, RG(D')) = \mathcal{M}_f(Q, D')$. Hence, it follows that $\mathcal{M}_f(Q, D) = \mathcal{M}_f(Q, D')$. That is, $D$ and $D'$ are equivalent.

For the other direction, consider the case where $D$ and $D'$ are equivalent. We start by showing that the set of labeled edges of $RG(D)$ is contained in the set of label edges of $RG(D')$.

Let $o_1o_2$ be an edge of $RG(D)$. According to the definition of the reachability graph (Definition 3.54), in $D$ there is an edge that enters $o_2$ and has the label $l$. In addition, $D$ either has a path from $o_1$ to $o_2$ or vice-versa. There are two cases to examine, depending on whether $o_1$ is the root of $D$ or not.

If $o_1$ is the root of $D$, then we create a query $Q_1$ as follows. The variables of $Q_1$ are $r_{Q_1}$ and $v_2$. There is a single edge in $Q_1$—an edge from $r_{Q_1}$ to $v_2$ with label $l$. The matching $\mu_1$ that maps $r_{Q_1}$ to $o_1$ and $v_2$ to $o_2$ is a flexible matching of $Q_1$ w.r.t. $D$. Since $D'$ is equivalent to $D$, $\mu_1$ is also a flexible matching of $Q_1$ w.r.t. $D'$. This means that $\mu_1$ satisfies the qec $lv_2$. Thus, in $D'$ there is an edge with label $l$ that enters $o_2$. In addition, since $o_1$ is the root of $D$ it is also the root of $D'$. Thus,
there is a path in $D'$ from $o_1$ to $o_2$, because all the nodes are reachable from the root. Consequently, $RG(D')$ has an edge $o_1 o_2$.

If $o_1$ is not the root of $D$, then we create a query $Q_2$ as follows. There are three variables in $Q_2$: $r_{Q_2}$, $v_1$ and $v_2$. The edges of $Q_2$ are the two edge $r_{Q_2} l_1 v_1$ and $v_1 l v_2$, where $l_1$ is a label on an edge that enters $v_1$. Note that such a label exists because there is a path in $D$ from the root to $o_1$. Let $\mu_2$ be a matching that maps $r_{Q_2}$ to the database root, $v_1$ to $o_1$ and $v_2$ to $o_2$. Obviously, $\mu_2$ is a flexible matching of $Q_2$ w.r.t. $D$. Since $D'$ is equivalent to $D$, $\mu_2$ is also a flexible matching of $Q_2$ w.r.t. $D'$. It means that $\mu_2$ satisfies the wec $v_1 l v_2$ and the qec $l v_2$ w.r.t. $D'$. That is, in $D'$ there is either a path from $o_1$ to $o_2$ or vice-versa. In addition, there is an edge with the label $l$ that enters $o_2$. Hence, $RG(D')$ has an edge $o_1 o_2$.

Thus far, we have shown that the set of edges of $RG(D)$ is contained in the set of edges of $RG(D')$. By exchanging the roles of $D$ and $D'$, we can show that the set of edges of $RG(D')$ is contained in the set of edges of $RG(D)$. This shows that $RG(D)$ and $RG(D')$ are isomorphic.

It is not necessary to solve the general case of graph isomorphism in order to decide equivalence of databases under the flexible semantics. Instead, it is sufficient to check that the identity mapping is an isomorphism. Thus, deciding whether two databases $D$ and $D'$ are equivalent under the flexible semantics is in $O(|D|^2 \log |D| + |D'|^2 \log |D'|)$.

### 3.5.1 Removing Redundancies

One application of database equivalence is the removal of redundant parts from databases. A part of a database is redundant if removing it from the database has no effect on the result of query evaluation.

For example, given a database $D$, we say that an edge $o_1 l o_2$ in $D$ is redundant w.r.t. the semiflexible (or flexible) semantics if $D$ has a path of the form $o_1 l o_2$ that does not include the edge $o_1 l o_2$. If a redundant edge is removed, the result is a database that is equivalent to the original one under the semiflexible (flexible) semantics.
Chapter 3 Flexible and Semiflexible Queries

Proposition 3.68 (Redundant Edges) Suppose that $D'$ is a database that is created from another database $D$ by removing from $D$ a redundant edge. Then $D$ and $D'$ are equivalent under both the semiflexible and the flexible semantics.

Proof. Equivalence under the semiflexible semantics follows from Theorem 3.65. This is because a removal of a redundant edge has no effect on semiflexible path inclusion nor on the strongly connected components.

Equivalence under the flexible semantics follows from Theorem 3.67. This is because a removal of a redundant edge cause no change in the reachability graph of the database.

The next lemma shows that under the semiflexible semantics, when there are no redundant edges, the inclusion condition for database equivalence can be relaxed. That is, if two databases are equivalent then each path, from the root to a leaf, in one database has a path, in the other database, that contains exactly the same set of objects and the same set of labels.

Lemma 3.69 (Image Path Permutation) Let $D$ and $D'$ be two databases over the same set of objects, with the same root and without any redundant edges. If $D$ and $D'$ are equivalent under the semiflexible semantics then for each path $o_0l_1o_1\cdots l_no_n$, from the root of $D$ to a leaf, there exists a permutation $\sigma$ of $1, \ldots, n$, such that in $D'$ there is a path $o_0\sigma_0l_1\sigma_1\cdots l_n\sigma_n$.

Proof. Let $\phi = o_0l_1o_1\cdots l_no_n$ be a path in $D$ such that in $D'$ no path has the form $o_0l_1\sigma_1\cdots l_n\sigma_n$. We show that this leads to a contradiction.

According to Theorem 3.65, there is a semiflexible path inclusion of $D$ in $D'$ and vice-versa. So, there exists a permutation $\sigma'$ such that $D'$ has a path $\phi' = o_0l_1\sigma'_{(1)}o_{\sigma'_{(1)}}\cdots l_n\sigma'_{(n)}$. According to our assumption, $\phi'$ contains a node $o'$ that is not among the nodes $o_0, o_{\sigma'_{(1)}}, \ldots, o_{\sigma'_{(n)}}$. Note that if $\phi'$ is not a simple path then $o'$ could be one of the nodes $o_1, \ldots, o_n$. Since there is a semiflexible path inclusion of $D'$ in $D$ we have a path $\hat{\phi}$ in $D$ that includes $\phi'$. The path $\hat{\phi}$ goes through all the nodes of $\phi$ and through the node $o'$ (it may, in addition, go through other nodes). Since $o_0$ is the root and $o_n$ is a leaf, the node $o'$ is neither the first nor
the last node in \( \hat{\phi} \). Therefore, there must be two nodes \( o_i \) and \( o_{i+1} \) in \( D \) such that in addition to the edge \( o_i l_{i+1} o_{i+1} \) there is a path \( o_i o'_{i+1} l_{i+1} o_{i+1} \). The edge \( o_i l_{i+1} o_{i+1} \) is redundant, in contradiction to the our assumption that \( D \) and \( D' \) are redundancy free. \( \square \)

### 3.5.2 Transforming a Database to an Equivalent Tree

In addition to testing database equivalence, it is also possible to transform a database \( D \) to a tree, provided that \( D \) is indeed equivalent to some tree database. Note that under the semiflexible semantics, a cyclic database cannot be equivalent to a tree database. However, there are dag databases that are equivalent to tree databases. Under the flexible semantics, even a cyclic database could be equivalent to a tree database.

Transforming a database to an equivalent tree is important for several reasons. First, evaluation of queries is more efficient when the database is a tree, as was shown earlier. Secondly, in a graphical user interface, presenting trees is easier than presenting dags. Thirdly, storing data as a document (e.g., in XML) is easier when the data is a tree, since references and identifiers need not be used.

**Definition 3.70 (Tree Transformable Database)** Given a database \( D \), we say that \( D \) is tree transformable under the semantics \( s \) if there exists a tree database \( T \) that is equivalent to \( D \) under the semantics \( s \).

In this section, we characterize tree transformable databases under the semiflexible and the flexible semantics. In addition, we give algorithms for the transformations.

**Tree Transformable Databases under the Semiflexible Semantics**

We discuss transformation to a tree under the semiflexible semantics. Given a database \( D \) that has no redundant edges, the following are necessary conditions for \( D \) to be tree transformable under the semiflexible semantics.

The first necessary condition is that \( D \) does not include a node that has incoming edges with different labels. The reason for this condition is that in a tree database
only one edge enters each node (except for the root that has no incoming edge) and this edge holds exactly one label. The second necessary condition is that $D$ is acyclic. The reason for the second condition is that in a finite tree all paths are finite, while a cyclic graph contains an infinite path. The third necessary condition is that the number of root-to-leaf paths in $D$ is not greater than the number of nodes in $D$. Recall that in a tree, the number of paths, from the root to a leaf, never exceeds the number of nodes.

In order to have conditions that are both necessary and sufficient, we need additional definitions. A path-nodes set of a database $D$ is a set of all the nodes on a path in $D$ from the root to a leaf. For a given database, the path hypergraph is a hypergraph whose hyperedges are path-nodes sets.

**Definition 3.71 (Path Hypergraph)** Given a database $D$, the path hypergraph of $D$ is the hypergraph $\mathcal{H}$ such that:

1. The nodes of $\mathcal{H}$ are the objects of $D$.

2. The hyperedges of $\mathcal{H}$ are the path-nodes sets of $D$.

**Definition 3.72 (Bachman Diagram)** Let $\mathcal{E}$ be the set that contains all the hyperedges of $\mathcal{H}$ and all the nonempty intersections of two or more hyperedges of $\mathcal{H}$. The Bachman diagram of $\mathcal{H}$, denoted $BD(\mathcal{H})$, is the following graph.

1. For each element of $\mathcal{E}$, there is a node in $BD(\mathcal{H})$.

2. There is an edge between $E_1$ and $E_2$ in $BD(\mathcal{H})$ if

   (a) $E_1 \subseteq E_2$, and

   (b) there is no element $E'$ in $\mathcal{E}$, such that $E_1 \subseteq E' \subseteq E_2$ and $E'$ is different from both $E_1$ and $E_2$.

We say that $BD(\mathcal{H})$ is acyclic if it contains no cycle (as an undirected graph). In Appendix A we provide a definition of a $\gamma$-acyclic hypergraph. A discussion of acyclic Bachman diagrams and their usage in the characterization of $\gamma$-acyclic hypergraphs is given in [31]. In [55], Bachman diagrams are used to characterize full disjunctions.
**Lemma 3.73 (Acyclicity is a Necessary Condition)** Let $D$ be a database with no redundant edges and $\mathcal{H}$ be the path hypergraph of $D$. If $D$ is tree transformable under the semiflexible semantics then $BD(\mathcal{H})$, the Bachman diagram of $\mathcal{H}$, is acyclic.

Proof. To prove the lemma, we assume that the Bachman diagram of $\mathcal{H}$ is cyclic and we show that $D$ is not tree transformable.

In [31], Fagin showed that for a hypergraph $\mathcal{H}$, the Bachman diagram of $\mathcal{H}$ is acyclic if and only if $\mathcal{H}$ is $\gamma$-acyclic. We have assumed that the Bachman diagram of $\mathcal{H}$ is cyclic and, hence, $\mathcal{H}$ itself is not $\gamma$-acyclic. This means that $\mathcal{H}$ either has a pure cycle or a $\gamma$-3-cycle.

All the hyperedges of $\mathcal{H}$ contain the root of $D$. Thus, there cannot be a pure cycle in $\mathcal{H}$. So, $\mathcal{H}$ must have a $\gamma$-3-cycle. That is, $\mathcal{H}$ includes three hyperedges $X, Y, Z$ and three nodes $n_0, n_1, n_2$ such that:

1. $n_0$ is an element of all the three sets $X, Y$ and $Z$;
2. $n_1$ is an element of $X$ and $Z$ and not an element of $Y$; and
3. $n_2$ is an element of $Y$ and $Z$ and not an element of $X$.

If $n_0$ is not the root of $D$, we replace it with the root. This can be done since the root of $D$ is an element of every hyperedge.

To derive a contradiction we assume that $D$ is tree transformable, i.e., there is a tree database $T$ such that $T$ is equivalent to $D$ under the semiflexible semantics. One of the following three options must hold in $T$:

1. No path in $T$ goes through both $n_1$ and $n_2$.
2. There is a path in $T$ from $n_1$ to $n_2$.
3. There is a path in $T$ from $n_2$ to $n_1$.

In Case 1, the path in $D$ whose nodes are the nodes of $Z$ is not included in any path of $T$. This is because $Z$ contains both $n_1$ and $n_2$. In Case 2, the path $\phi_Y$ in $D$, whose nodes are the nodes of $Y$ includes $n_0$ and $n_2$ and does not include $n_1$. According to Lemma 3.69, $T$ should have a path that includes the same nodes as
\( \phi_Y \). However, such a path cannot exist, since \( T \) is a tree and every path in \( T \) that includes \( n_2 \) also include \( n_1 \). In Case 3, each path in \( T \) that includes \( n_1 \) must include \( n_2 \). Thus, there is no path in \( T \) that includes exactly the nodes of \( X \), i.e., includes \( n_0 \) and \( n_1 \) and does not include \( n_2 \). Since all three cases lead to a contradiction, an equivalent tree database of \( D \) does not exist.

Next we explain why the test of Lemma 3.73 can be performed in polynomial time in the size of the given database.

Consider a hypergraph \( H \) with \( n \) hyperedges. It is possible to check in polynomial time in the size of \( H \) that the Bachman diagram of \( H \) is acyclic. This is based on the following two observations. First, in an acyclic diagram, the number of paths from the least node (the node that is contained in all the other nodes) to the leaves (i.e., the hyperedges of \( H \)) is at most \( n \). Second, there are at most \( n \) sets in each path from the root to a leaf. To see why the second observation is true, consider two nodes in the diagram \( E_1 \) and \( E_2 \) that have an edge between them, where \( E_1 \subseteq E_2 \). Let \( S_1 \) (\( S_2 \)) be the set of hyperedges that \( E_1 \) (\( E_2 \)) is their intersection. Then \( S_1 \supset S_2 \).

Thus, there are at most \( n^2 \) nodes in the Bachman diagram of \( H \). It is possible to construct the diagram by creating all the intersection of hyperedges of \( H \). If at some point there are more than \( n^2 \) nodes then the test stops with the answer that the diagram is cyclic. After constructing the diagram, it is possible to check in linear time, in the size of the diagram, whether there are cycles. This can be done using a DFS traversal.

The test of Lemma 3.73 can be performed in polynomial time in the size of the given database. This is based on the observation that in a tree database the number of paths cannot exceed the number of nodes. In the test, we construct all the hyperedges of the path hypergraph. If at some point, during the construction, the number of hyperedges exceeds the number of database nodes, then the test stops with the answer that the database is not tree transformable. Otherwise, the Bachman diagram is constructed and tested for acyclicity.

Thus far, we discussed necessary conditions for the existence of an equivalent
Chapter 3  Flexible and Semiflexible Queries

tree database. Next, we show that the conditions that were presented as necessary
conditions are also sufficient conditions for performing a transformation. In addition,
we present a transformation algorithm.

We say that a database $D$ is a tree-transformable candidate if (1) $D$ has no
redundant edges; (2) $D$ is acyclic; (3) there is no node in $D$ that has two different
incoming labels; and (4) the Bachman diagram of the path hypergraph of $D$ is
acyclic. The algorithm that we present next can transform a tree-transformable
candidate to an equivalent tree. Thus, it shows that tree-transformable candidates
are actually tree transformable.

Figure 3.4 presents a polynomial-time algorithm that transforms a given database
$D$ to a tree, provided that $D$ is tree transformable. The algorithm starts by creating
the path hypergraph $\mathcal{H}$ of $D$. Note that in a tree transformable database, the
number of paths from the root to the leaves cannot exceeds the number of nodes,
and thus, the size of the path hypergraph is polynomial in the size of the database.

The second step of the algorithm is the creation of the Bachman diagram $B =
BD(\mathcal{H})$. This can be done in polynomial time in the size of $\mathcal{H}$. Recall that the nodes
of $B$ are sets of objects of the database $D$. The intersection of all the nodes of $B$
always contains the root of $D$. Thus, there is a least node of $B$ that is contained in
all the other nodes. In the algorithm, we view $B$ as a tree whose root is the least
node.

The algorithm creates a tree database $T$ by visiting the nodes of $B$ in a topological
order, starting with the least node. Initially, $T$ is empty. For each visited node $E$ of
$B$, the algorithm adds to $T$ the objects of $E - E'$, where $E'$ is the parent of $E$. The
newly added objects are connected by new edges to form a simple path. The order
of the objects along this path is not important, except in the case of the least node
of $B$, since the root of $D$ must be the first object on the path created for the least
node. Thus, each node $E$ of $B$ is associated with some path in $T$, and this path has
a first object and a last object. A new edge is also added from the last object of the
parent of $E$ to the first object of $E - E'$.

The final step of the algorithm is to label the edges of $T$. Each edge is labeled
with the unique label associated with the object that it enters; that is, if an edge
Algorithm TransformingDatabaseToTree(D);

**Input** a tree-transformable database $D$;

**Output** a tree database $T$ that is equivalent to $D$
under the semiflexible semantics;

let $\mathcal{H}$ be the path hypergraph of $D$;

let $B = BD(\mathcal{H})$ be the Bachman diagram of $\mathcal{H}$;

$S \leftarrow \emptyset$;

let $S_0$ be the least node of $B$ ($S_0$ is the intersection of all nodes of $B$);

(* by choosing $S_0$ to be the root, $B$ can be viewed as a tree *)

create from the objects of $S_0$ a path $P_0$ with $r_D$ (the root of $D$) as the first node;

add to $S$ all the children of $S_0$ in $B$;

while $S$ is not empty

remove some node $S_j$ from $S$;

add to $S$ all the children of $S_j$ in $B$;

let $S_i$ be the parent of $S_j$ in $B$;

create a simple path $P_j$ from the objects of $S_j$ that are not in $S_i$;

(* Since $S_i$ is the parent of $S_j$, a path $P_i$ has already been created for $S_i$ *)

add an edge from the last node of $P_i$ to the first node of $P_j$;

let $T$ be the database that was produced in the previous steps;

label $T$ by attaching to each edge the label that corresponds to its target in $D$;

return $T$;

---

Figure 3.4: Creating a tree database $T$ from a database $D$.

enters an object $o$, then in the original database $D$, all the edges that enter $o$ are labeled with the same label $l$, and $l$ is also the label of the single edge that enters object $o$ in the tree database $T$. Note that the root has no incoming edges in $T$. It is easy to see that the returned graph is indeed a tree.

**Lemma 3.74 (Correctness of the Transformation Algorithm)** If a database
Chapter 3  Flexible and Semiflexible Queries

D is a tree-transformable candidate, then applying the algorithm Transforming-DatabaseToTree to D produces a tree that is equivalent to D under the semiflexible semantics. Furthermore, the runtime of the transformation is polynomial in the size of D.

Proof. Let T be the graph that the algorithm returns. We claim that T contains exactly the same set of objects as D. Note that the creation of T is done by traversing the Bachman diagram B and visiting all the nodes in B. Since B is connected and has no cycles, each node is visited exactly once. In the traversal, an object is added to T if it is an element of a node S_j and not an element of the parent of S_j. It is easy to see that all the objects of T are objects of D. This is because the nodes of B are sets of objects of D. Secondly, every node of D appears in at least one node of B and thus appears in T. Thirdly, none of the objects is added twice to T. To see this, consider the case where an object o is added to T twice. Let S_j and S_k be the nodes of B for which o was added to T. Then the intersection of S_j and S_k is non-empty and contains o. This means that in B there is an ancestor S_i of S_j and S_k and S_i contains o. Hence, the parent of S_j contains o in contradiction to S_j being a node for which o was added to T. The graph T is a tree because the construction of T is in such a way that the root has no incoming edge and every object, other than the root, has exactly one incoming edge. In addition, due to the way edges receive their labels, an object in T has an incoming edge with the label l if and only if it has an incoming edge in D with label l.

Next, we show that D and T are equivalent under the semiflexible semantics. The equivalence is proved by showing that there is a semiflexible path inclusion in both directions, according to Theorem 3.65.

We start by showing a semiflexible path inclusion of D in T. Consider a path \( \phi = o_0l_1o_1 \cdots l_no_n \) in D. We assume that \( \phi \) is a path from the root to a leaf. This assumption can be done because every path is contained in a path from the root to a leaf, so inclusion can be shown w.r.t. the container. The path \( \phi \) is represented by an hyperedge in H and thus B has a node S that contains all the objects on
Chapter 3  Flexible and Semiflexible Queries

φ. Let $S_0, S_1, \ldots, S_k$ be a path in $B$, such that $S_0$ is the least node and $S_k$ is $S$. The algorithm creates a path from the objects of each node $S_i$ and connects the last node on each path to the first node of the next path. The result is a path $φ'$ in $T$ that contains precisely the objects of $S_k$. All the objects of $φ$ are included in $φ'$. In addition, according to the way labels are given, for each $1 \leq i \leq n$, the label on the edge that enters $o_i$, in $φ'$, is $l_i$. Hence, there is a semiflexible path inclusion of $φ$ in $φ'$.

To show inclusion in the opposite direction, consider a path $φ' = o_0l_1o_1 \cdots l_no_n$ in $T$, from the root to a leaf. Let $S_i$ be the node in $B$ that contains $o_n$. Obviously, $S_i$ is a leaf in $B$ and the set of objects $S_i$ is equal to the set of objects on $φ'$. Since $S_i$ has no children it is not an intersection of two other nodes in $B$. Thus, $S_i$ is an hyperedge of $H$. This means that there is a path $φ$ in $D$ such that $φ$ is a path from the root to a leaf and the set of nodes on $φ$ is equal to $S_i$. The two paths $φ$ and $φ'$ consists of the same objects. Furthermore, if $l$ is the label on the edge that enter $o$ in $φ'$ then $l$ is also the label on the edge that enters $o$ in $φ$. To conclude, there is a semiflexible path inclusion of $φ'$ in $φ$.

In the algorithm, each node of $B$ is traversed exactly once. For each node, a single pass over the objects of the node is needed since the algorithm does not require to connect the objects in any particular order. Thus, the construction of $T$ is linear in the size of $B$. The number of leaves in $B$ is at most the number of paths, from the root to a leaf, in $D$. If $D$ is tree transformable, it cannot have more paths, from the root to a leaf, than nodes. Hence, the size of $B$ is linear in the size of $D$. Finally, since the construction of $B$ is polynomial in the size of $D$, the algorithm has a polynomial runtime in the size of $D$. \[\square\]

We conclude the discussion on tree transformable databases, under the semiflexible semantics, with the following theorem.

**Theorem 3.75 (Semiflexible Equivalence to a Tree)** Let $D$ be a database with no redundant edges and with the path hypergraph $H$. The database $D$ is tree transformable under the semiflexible semantics if and only if the following three conditions hold.

Chapter 3  Flexible and Seflexible Queries

Figure 3.5: Transformation to a tree under the semiflexible semantics.

1. $D$ is acyclic;

2. There is no node in $D$ that has two different incoming labels; and

3. BD$(H)$, the Bachman diagram of $H$, is acyclic.

Deciding if $D$ is tree transformable under the semiflexible semantics is in polynomial runtime in the size of $D$.

Proof. Correctness of the theorem follows from Lemma 3.73 and Lemma 3.74. □

Example 3.76 In Figure 3.5, a tree transformable database $D$ is depicted. In addition, the Bachman diagram of $D$ and an equivalent tree database of $D$ are shown. When applying the transformation algorithm to $D$, the result is either the shown tree database or a tree database that is equivalent to the one that is shown.

Tree Transformable Databases under the Flexible Semantics

We now consider transformation to a tree under the flexible semantics. A necessary condition for the existence of a transformation under the flexible semantics is as follows. The database should not have a node $o$, such that two edges with different labels enter $o$. Moreover, there should not be any edge that enters the root.

Next, we give some additional definitions. The augmented reachability graph of a database $D$ is the reachability graph of $D$ augmented with an unlabeled edge from
each node to the root. An edge between two nodes, in the augmented reachability graph, reflects the existence of a path in $D$ between these nodes. Since each node in the database is reachable from the root, there is an edge in the augmented reachability graph from each node to the root.

The maximal-clique hypergraph of a database $D$ is the hypergraph that has the same nodes as $D$ and has, as hyperedges, the maximal cliques in the augmented reachability graph of $D$. Note that in a directed graph, a clique is a set of nodes, such that every two nodes are connected by edges in both directions. A clique is maximal if it is not contained in any other clique.

Given that a database $D$ is tree transformable under the flexible semantics, the creation of the equivalent tree $T$ is the same as in the case of the semiflexible semantics, except for the following. In the algorithm of the transformation (Figure 3.4), the hypergraph $H$ that is used is the maximal-clique hypergraph of the augmented reachability graph of $D$ (instead of the path hypergraph of $D$).

**Theorem 3.77 (Flexible Equivalence to a Tree)** Consider a database $D$. Let $H$ be the maximal-clique hypergraph that is created from the augmented reachability graph of $D$. The database $D$ is tree transformable under the flexible semantics if and only if the following three conditions hold.

1. If there is an edge in $D$ that enters the root, then the label on this edge is equal to the labels on all the other edges of $D$;
2. There is no node in $D$ that has two different incoming labels; and
3. The Bachman diagram $BD(H)$ of $H$ is acyclic.

**Proof.** We start by showing that the conditions are necessary. The first condition is necessary because in a tree no edge enters the root and only a single edge enters each other node. Thus, in the reachability graph of a tree, the root and every other node are connected by a single edge and this edge carries a single label. Yet, if in $D$ there is an edge labeled with $l$ that enters the root $r_D$ while an edge labeled with $l'$ enters a different node $o$, then in the reachability graph there should be two edges between $r_D$ and $o$—one that is labeled with $l$ and another that is labeled with $l'$. 
Chapter 3 Flexible and Semiflexible Queries

The necessity of the second condition is obvious—a node in a tree cannot have two incoming edges and thus cannot have two incoming labels.

The proof for the necessity of the third condition is similar to the proof of Lemma 3.73—we assume that $BD(\mathcal{H})$ contains a cycle and show that $D$ has no equivalent tree, under the flexible semantics.

If $BD(\mathcal{H})$ contains a cycle then $\mathcal{H}$ is not $\gamma$-acyclic. Since all the hyperedges of $\mathcal{H}$ contain the root of $D$ there is no pure cycle in $\mathcal{H}$. Thus, $\mathcal{H}$ must have a $\gamma$-3-cycle. The $\gamma$-3-cycle is created by three hyperedges $X$, $Y$ and $Z$ and three nodes $r_D$, $n_1$ and $n_2$ such that (1) $r_D$ is an element of all the three hyperedges; (2) $n_1$ is an element of $X$ and $Y$ and not an element of $Z$; and (3) $n_2$ is an element of $X$ and $Z$ and not an element of $Y$.

Consider a tree $T$ that is equivalent to $D$ under the flexible semantics. From the equivalence it follows that the reachability graph of $T$ should be isomorphic to the reachability graph of $D$. We examine three possible cases:

1. There is no path in $T$ that contains both $n_1$ and $n_2$.
2. In $T$ there is a path from $n_1$ to $n_2$.
3. In $T$ there is a path from $n_2$ to $n_1$.

If in $T$ there is no path that contains both $n_1$ and $n_2$, then there is no edge between $n_1$ and $n_2$ in the reachability graph of $T$. This means that there is no edge between $n_1$ and $n_2$ in the reachability graph of $D$, in contradiction to having the hyperedge $X$ in $\mathcal{H}$, since the nodes in $X$ are a clique.

If in $T$ there is a path from $n_1$ to $n_2$, then every path that goes through both the root and $n_2$ must also go through $n_1$. However, according to the hyperedge $Z$ there is a clique in the augmented reachability graph of $D$ (and thus also in the augmented reachability graph of $T$) that contains $n_2$ and does not contain $n_1$. This situation requires the existence of a node $o$ in $T$, such that there is a path that goes through both $o$ and $n_2$ and there is no path that goes through both $o$ and $n_1$. However, since $T$ is a tree no such object $o$ can exist.

For the third case, where in $T$ there is a path from $n_2$ to $n_1$, showing that it is impossible is done similarly to the second case above.
Chapter 3  Flexible and Semiflexible Queries

Figure 3.6: Transformation to a tree under the flexible semantics.

To prove that the conditions are sufficient, we need to show that when the conditions hold, for a database $D$, the transformation algorithm returns a tree database that is equivalent to $D$ under the flexible semantics. This is shown by showing that the reachability graph of $D$ is isomorphic to the reachability graph of $T$, where $T$ is returned by the algorithm. Since all the labels that enter a node $o$ in $D$ are equal to the labels on the edge that enters $o$ in $T$, it is sufficient to show that a pair of nodes in $D$ is connected by a path if and only if these two nodes are connected by a path in $T$.

If there are two nodes $o_1$ and $o_2$ such that $D$ has a path from $o_1$ to $o_2$, then there is a maximal clique in the augmented reachability graph of $D$ that contains both $o_1$ and $o_2$. It follows that $H$ contains a node such that $o_1$ and $o_2$ are elements of this node and thus, in $T$, there is either a path from $o_1$ to $o_2$ or vice-versa.

For the other direction, if there are two nodes $o_1$ and $o_2$ such that in $T$ there is path from $o_1$ to $o_2$, then there is a leaf of $BD(H)$ that contains both $o_1$ and $o_2$. This means that there is a maximal clique in $H$ that contains both $o_1$ and $o_2$. Thus, in $D$ there is either a path from $o_1$ to $o_2$ or vice-versa.

The creation of the maximal-clique hypergraph is not in polynomial time. Thus, the transformation to a tree under the flexible semantics is not in polynomial runtime.

**Example 3.78** An example of a transformation is depicted in Figure 3.6. Note that Figure 3.6 does not show the augmented reachability graph of the database
D. However, it is easy to see that the augmented reachability graph contains two maximal cliques. One clique is the set \( \{ o_0, o_1, o_3, o_4 \} \) and the other clique is the set \( \{ o_0, o_1, o_2, o_3, o_5 \} \). These cliques are the leaves of the Bachman diagram that is created from the maximal-clique hypergraph of the augmented reachability graph.

3.6 Summary of Contributions

In this chapter, we introduced novel query semantics, namely, the semiflexible and the flexible semantics. These semantics facilitate querying of data when the schema is unknown, complicated or changes frequently. Meaningful queries can be formulated even when the user is oblivious to the structural details of the data and is only familiar with the ontology. Moreover, queries are insensitive to common variations in the schemas of semantically similar data instances.

We investigated query evaluation under the new semantics. Under the semiflexible semantics, a polynomial-time algorithm, under input-output complexity, was introduced for the case of a DAG query and a tree database, provided that labels are not repeated in query paths. The case where the database is a tree is important since XML documents are usually trees. It is also shown that when the database is cyclic, there is no polynomial-time algorithm, in the size of the input and the output, assuming that \( P \neq NP \). This last claim holds even when the query is a path.

Under the flexible semantics, evaluation has the same complexity as under the conventional rigid semantics. Thus, for a tree query, evaluation can be done in polynomial time, under input-output complexity. For DAG queries, there is no polynomial time algorithm, in the size of the input and the output, assuming that \( P \neq NP \).

We characterized query containment and query equivalence, under both the semiflexible and the flexible semantics. The provided characterizations are a first step towards developing optimization techniques. It is shown that equivalence of queries is decidable in polynomial time.
Chapter 3  Flexible and Semiflexible Queries

An additional contribution presented in this chapter is in introducing and investigating the novel concept of database equivalence. Under the conventional rigid semantics, two database are equivalent only if they are isomorphic. This is not the case under the semiflexible and the flexible semantics. Database equivalence is defined and characterized, under the two semantics.

A novel feature of the new semantics is the possibility of transforming a given database $D$ to a tree database that is equivalent to $D$. For the semiflexible semantics, testing whether $D$ is equivalent to a tree database and actually transforming $D$ to a tree database (if the test is positive) are both polynomial. Since queries can be evaluated more efficiently over tree databases, this result is of practical importance. In addition, the transformation allows to represent data in XML without using references. This is important since, for some parsers, references are problematic.
Chapter 4

Queries with Maximal Answers

This chapter introduces two new query semantics. The new semantics facilitate querying of incomplete data. The outline of the chapter is as follows. Section 4.1 describes two query semantics that allow maximal rather than complete answers, namely the or-semantics and the weak semantics. In Section 4.2, an algorithm for computing maximal or-matchings is presented, along with a proof of correctness and an analysis of the time complexity. In addition, it is shown how to modify the algorithm in order to compute maximal weak matchings. Finally, in Section 4.3, we discuss the contribution of this chapter.

4.1 Query Semantics

In the rigid semantics, matchings are complete assignments, i.e., all the query variables are mapped to database objects. We now define matchings that are partial assignments, i.e., some query variables may not be mapped to a database object and, instead, are assigned a null value.

4.1.1 Partial Matchings

We start with a formal definition of partial matchings.

Definition 4.1 (Partial Assignment) Let $Q = (V, E_Q, r_Q)$ be a query and $D = (O, E_D, r_D, \alpha)$ be a database. A partial assignment of $Q$ w.r.t. the database $D$ is a mapping $\mu : V \rightarrow O \cup \{\bot\}$, such that each variable of $Q$ is mapped either to an object of $D$ or to a special value that is called null and is denoted as $\bot$. 
Chapter 4 Queries with Maximal Answers

Figure 4.1: Another fragment of the movie database.

Each partial assignment can be represented as a graph. In the graph the nodes are pairs consisting of a variable and an object. Each pair denotes a mapping of an object to a variable. This way of representing partial assignments is formally specified in the next definition.

Definition 4.2 (Assignment Graph) An assignment graph $G$ of a query $Q = (V, E_Q, r_Q)$ w.r.t. a database $D = (O, E_D, r_D, \alpha)$ consists of nodes of the form $(v, o)$, where $v$ is a variable of $Q$ and $o$ is an object of $D$. The assignment graph $G$ cannot have any repeated variable, i.e., it cannot have any variable $v$ that appears in two distinct nodes $(v, o_1)$ and $(v, o_2)$, where $o_1 \neq o_2$. The assignment graph $G$ may (but does not necessarily) have an edge from $(v_1, o_1)$ to $(v_2, o_2)$, denoted $(v_1, o_1)l(v_2, o_2)$, provided that $v_1lv_2$ is an edge of $Q$ and $o_1lo_2$ is an edge of $D$. We say that $(v_1, o_1)l(v_2, o_2)$ is a potential edge of $G$ if $v_1lv_2$ is in $Q$ and $o_1lo_2$ is in $D$. If the edge $(v_1, o_1)l(v_2, o_2)$ appears in $G$, then it is called an actual edge of $G$.

An assignment graph $G$ corresponds to an assignment $\mu$ that is defined as follows. If $(v, o)$ is a node of $G$, then $\mu(v) = o$; and if $G$ has no node with $v$ as its first component (i.e., no node of the form $(v, o)$ for some $o$), then $\mu(v) = \bot$. The assignment $\mu$ is well defined, since $G$ does not have any repeated variable.

Conversely, if $\mu$ is an assignment, then an assignment graph for $\mu$ comprises all nodes of the form $(v, \mu(v))$, where $\mu(v)$ is non-null. The assignment $\mu$ may have
several assignment graphs. All those graphs have the same set of nodes, but they may have different sets of edges. However, the set of actual edges is always a subset of the set of potential edges.

Let $G$ be an assignment graph of a query $Q$ with respect to a database $D$. The closure of $G$ is obtained by adding to $G$ all potential edges that are not already in $G$. We say that $G$ is closed if $G$ is equal to its closure.

A matching is an assignment that satisfies some additional conditions (recall Definition 2.4). In this chapter, we define two types of matchings. The definitions are equivalent to those in [41]. Note that [41] presents three semantics for queries with maximal answers. We discuss only two of the three semantics, since in the semantics that is not presented here, evaluation of cyclic queries is NP-hard.

**Definition 4.3 (OR-Matching)** An assignment $\mu$ of a query $Q = (V, E_Q, r_Q)$ w.r.t. a database $D = (O, E_D, r_D, \alpha)$ is an OR-matching if $\mu$ has an assignment graph $M$ that satisfies the following conditions.

1. The graph $M$ has the node $(r_Q, r_D)$ (i.e., $\mu(r_P) = r_D$) and this node is designated as the root of $M$.

2. Each node of $M$ is reachable from the root.

An assignment graph $M$ of a query $Q$ w.r.t. a database $D$ preserves the edges of $Q$ if it satisfies the following condition. For every two nodes $(v_1, o_1)$ and $(v_2, o_2)$ of $M$, if $v_1 v_2$ is an edge of $Q$, then $(v_1, o_1)(v_2, o_2)$ is an edge of $M$ (and thus, $o_1 o_2$ is an edge of $D$).

**Definition 4.4 (Weak Matching)** An assignment $\mu$ is a weak matching of $Q$ if it has an assignment graph $M$ that satisfies the conditions of an OR-matching and, in addition, $M$ preserves the edges of $Q$.

An assignment graph that satisfies the conditions of an OR-matching is called an OR-matching graph. Similarly, an assignment graph that satisfies the conditions of a weak matching is called a weak-matching graph. Note that a weak-matching graph is always closed.
Essentially, the result of posing a query to a database is a set of matchings. Under the or-semantics, this set comprises all the or-matchings. Under the weak semantics, this set comprises all the weak matchings.

4.1.2 Subsumption and Maximal Matchings

Let \( \mathcal{A} \) be a set of assignments of a query \( Q \) w.r.t. a database \( D \). Given two assignments \( \mu \) and \( \mu' \) of \( \mathcal{A} \), we say that \( \mu \) subsumes \( \mu' \), denoted \( \mu' \subseteq \mu \), if \( \mu(x) = \mu'(x) \) whenever \( \mu'(x) \) is non-null. The assignment \( \mu \in \mathcal{A} \) is a maximal element of \( \mathcal{A} \) if no element in \( \mathcal{A} \), other than \( \mu \) itself, subsumes \( \mu \). Intuitively, if \( \mu' \subseteq \mu \), then \( \mu \) has more information than \( \mu' \). Therefore, all non-maximal elements can be discarded from \( \mathcal{A} \) without any loss of information.\(^\dagger\)

Subsumption is defined for assignment graphs in the natural way. Let \( G_1 \) and \( G_2 \) be assignment graphs. We say that \( G_1 \) subsumes \( G_2 \) if \( G_2 \) is a subgraph of \( G_1 \). That is, the set of nodes (edges) of \( G_2 \) is a subset or equal to of the set of nodes (edges) of \( G_1 \). An assignment graph is maximal, in a set of assignment graphs, if it is not subsumed by any other graph in the set.

Consider two assignment graphs \( G_1 \) and \( G_2 \) that correspond to the assignments \( \mu_1 \) and \( \mu_2 \), respectively. It is easy to see that if \( G_1 \) subsumes \( G_2 \) then \( \mu_1 \) subsumes \( \mu_2 \). Similarly, if \( \mathcal{M} \) is a set of assignment graphs and \( G \) is a maximal element of \( \mathcal{M} \), then the assignment that corresponds to \( G \) is maximal among the assignments that correspond to the elements of \( \mathcal{M} \).

Given a query \( Q \) and a database \( D \), the set of all maximal or-matchings of \( Q \)

\(^\dagger\)This may not be true if bag semantics has to be assumed (e.g., in order to evaluate aggregate queries). However, the issue of bag semantics is beyond the scope of this work.
### Figure 4.3: $\mathcal{M}_{or}(Q,D)$ and $\mathcal{M}_{w}(Q,D)$ for the $Q$ and $D$ of Figures 4.1 and 4.2.

w.r.t. $D$ is denoted as $\mathcal{M}_{or}(Q,D)$. Similarly, the set of all maximal weak matchings of $Q$ w.r.t. $D$ is denoted as $\mathcal{M}_{w}(Q,D)$. Note that both $\mathcal{M}_{or}(Q,D)$ and $\mathcal{M}_{w}(Q,D)$ can be viewed as relations with columns that are labeled with the variables of $Q$ and tuples that are filled with oid’s and nulls.

### Example 4.5
Figure 4.2 shows a query over the movie database of Figure 4.1. Intuitively, the query looks for an actor (along with her name and date of birth) and a movie (along with its title and language). The query specifies two relationships between the movie and the actor. First, the edge labeled with “filmography item” requires that the actor indeed acted in the movie. Secondly, the edge labeled with “director” requires that the actor was also the director of the movie.

The answers to the query differ according to the semantics that is used. The set of maximal or-matchings and the set of maximal weak matchings are shown in Figure 4.3. Each matching is shown as a tuple of oid’s and nulls. For atomic nodes, the value is shown next to the oid, inside parenthesis. Formally, however, a matching is always an assignment of oid’s and not of values.

In a maximal weak matching, the edges of the query are preserved. Thus, an answer to the query includes a movie and an actor only if the actor acted in the movie and was also the director of the movie. The first weak matching in Figure 4.3 finds Woody Allen as both an actor and the director of the movie Zelig. An actor will be in an answer without a movie (i.e., null in $v_2$) if none of the movies of the actor was directed by her. A movie will be in an answer without an actor (i.e., null in $v_3$) if none of the actors in the movie was also a director of the movie. The second weak matching in Figure 4.3 includes the movie Antz without an actor, since no

<table>
<thead>
<tr>
<th>Semantics</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5 (Zelig)</td>
<td>6 (English)</td>
<td>10 (Woody Allen)</td>
<td>11 (1/12/1935)</td>
</tr>
<tr>
<td>Weak</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5 (Zelig)</td>
<td>6 (English)</td>
<td>10 (Woody Allen)</td>
<td>11 (1/12/1935)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantics</th>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$v_3$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>8 (Antz)</td>
<td>$\bot$</td>
<td>10 (Woody Allen)</td>
<td>11 (1/12/1935)</td>
</tr>
<tr>
<td>Weak</td>
<td>1</td>
<td>3</td>
<td>$\bot$</td>
<td>8 (Antz)</td>
<td>$\bot$</td>
<td>$\bot$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>
actor in *Antz* was also the director of this movie.

In a maximal OR-matching, the edges of the query are not necessarily preserved. Thus, an OR-matching can include an actor and a movie even if the actor neither acted in the movie nor was the director of the movie. The second OR-matching in Figure 4.3 gives a movie and an actor, such that the actor acted in the movie but was not the director. There are no other maximal OR-matchings in this example, since a movie always has an actor and an actor (or a director) always has a movie in the given database.

### 4.2 Computing Maximal Matchings

Algorithms for computing the sets $\mathcal{M}_q(Q, D)$ and $\mathcal{M}_w(Q, D)$ were given in [42] for queries that are acyclic graphs. The time complexity of these algorithms was shown to be polynomial in the size of the input (i.e., the query and the database) and the output (i.e., the set of all maximal matchings). In this section, we generalize this result to queries that may have cycles.

#### 4.2.1 The Product Graph

We define a graph that combines the query and the database. This graph will be used for computing OR-matchings and weak matchings.

**Definition 4.6 (Product)** Consider a query $Q = (V, E_Q, r_Q)$ and a database $D = (O, E_D, r_D, \alpha)$. The product of $Q$ and $D$, denoted $Q \times D$, is the graph $\mathcal{P} = (V, \mathcal{E}, r_P)$, that satisfies the following conditions.

1. The root of $\mathcal{P}$, denoted $r_P$, is the pair $(r_Q, r_D)$;

2. The set of nodes of $\mathcal{P}$, denoted $\mathcal{V}$, consists of all nodes of the form $(v, o)$, where $v$ is a variable of $Q$ and $o$ is an object of $D$, i.e., $\mathcal{V} = V \times O$; and

3. The set of edges of $\mathcal{P}$, denoted $\mathcal{E}$, is a set that consists of all the edges of the form $(v_1, o_1)(v_2, o_2)$, where $v_1v_2$ is an edge of $Q$ and $o_1o_2$ is an edge of $D$. 
Figure 4.4: The product of the query that is presented in Figure 4.2 and the database that is presented in Figure 4.1.

Example 4.7 Figure 4.4 shows the product of the query that is presented in Figure 4.2 and the database that is depicted in Figure 4.1. Note that only nodes that are reachable from the root are shown.

The graph $G$ is a subgraph of $Q \times D$ if $G$ comprises some subset of the nodes of $Q \times D$. The edges of $G$ may be some or all the edges of $Q \times D$ that connect nodes appearing in $G$. It is rather obvious that if $M$ is either an or-matching graph or a weak-matching graph, then $M$ is a subgraph of $Q \times D$. The next proposition states when the converse is also true.

Proposition 4.8 Consider the following four conditions on a subgraph $G$ of $Q \times D$.

1. $G$ has no repeated variables; that is, $G$ does not have two nodes $(v, o_1)$ and $(v, o_2)$, such that $o_1 \neq o_2$.

2. $G$ contains the root of $Q \times D$.

3. In $G$, each node is reachable from the root.

4. $G$ preserves the edges of $Q$.

A subgraph $G$ of $Q \times D$ is an or-matching graph if and only if it satisfies the first three conditions. $G$ is a weak-matching graph if and only if it satisfies all four conditions.
Chapter 4  Queries with Maximal Answers

Proof. Condition 1 is needed so that $G$ will be the assignment graph of an assignment $\mu$ that is defined as follows. If $(v, o)$ is a node of $G$, then $\mu(v) = o$; otherwise, $\mu(v) = \bot$. Conditions 2 and 3 are identical to the two conditions in the definition of an or-matching. Condition 4 is the additional condition in the definition of a weak matching.

\[\Box\]

4.2.2 Computing Maximal OR-Matchings

In this section, we will show how to compute the maximal or-matchings of a query $Q$ w.r.t. a database $D$. The main idea is to compute all the maximal subgraphs $G$ of $Q \times D$, such that $G$ satisfies Conditions 1–3 of Proposition 4.8.

The computation is incremental; it starts with small subgraphs and expands them by adding edges. Initially, the root is the only current subgraph. In every iteration, each current subgraph is expanded by adding one more edge from $Q \times D$. The algorithm works efficiently if the edges are added in the “right order.” If $Q$ is a DAG (directed acyclic graph), a topological sort of $Q$ gives the right order and, in this case, there is really no need to consider the product $Q \times D$. Instead, it is sufficient to apply topological sort just to $Q$; see the details in [41, 42].

The General Idea

When $Q$ has cycles, a different approach is needed. Instead of $Q$ itself, the product $Q \times D$ should be considered. Furthermore, when adding edges in the “right order,” it is not sufficient to consider each edge just once. To formalize the notion of the “right order,” we say that an edge $m_1m_2$ of $Q \times D$ has a depth $d$ if there is a path consisting of $d$ edges that starts at the root of $Q \times D$ and ends with the edge $m_1m_2$. Two observations should be noted. First, an edge can have more than one depth when $Q \times D$ is not a tree. Second, an edge with depth 1 emanates from the root of $Q \times D$.

We divide the edges of $Q \times D$ into strata, such that stratum $k$ contains all the edges at depth $k$ (and $k$ is called the depth of the stratum). Note that an edge may belong to more than one stratum. A stratum traversal of $Q \times D$ is an ordered list $T$
that starts with the edges of stratum 1, followed by the edges of stratum 2, and so on. Notice that the order of the edges within each stratum is unimportant and can be determined arbitrarily, but the edges of stratum \( k \) must appear before the edges of stratum \( k + 1 \). Clearly, a stratum traversal may have multiple occurrences of the same edge, but in each stratum a given edge may appear at most once.

An important part of the algorithm for computing maximal OR-matchings is to create a stratum traversal \( T \) in polynomial time. However, as defined, a stratum traversal could be infinite. We will now show how to construct a finite stratum traversal \( T \) in polynomial time.

An occurrence of an edge \( e \) in stratum \( k \) is \textit{extraneous} if \( e \) is not the \( k \)th edge of any path that emanates from the root and consists of distinct edges. In principle, all extraneous occurrences of edges can be removed from a stratum traversal, but doing so is computationally hard. We will use a stratum traversal with some extraneous occurrences of edges. However, these extraneous occurrences neither affect the correctness of the algorithm nor add an exponential blowup to the time complexity.

Obviously, if there are \( m \) edges in \( Q \times D \), then only the first \( m \) strata may have non-extraneous occurrences of edges. Actually, it is sufficient for a stratum traversal to include only the first \( n \) strata, where \( n \) is the number of nodes in \( Q \). The reason for this is that a matching graph cannot include two nodes \((v, o_1)\) and \((v, o_2)\), such that \( o_1 \neq o_2 \). Therefore, the number of nodes in a matching graph cannot exceed the number of nodes in \( Q \).

The algorithm for computing maximal OR-matchings uses a stratum traversal \( T \) that consists of the first \( n \) strata, where \( n \) is the number of nodes in the query \( Q \), and is constructed as follows. Stratum 1 contains all the edges that emanate from the root of \( Q \times D \). Stratum \( k \) contains all edges \((v_1, o_1)l_1(v_2, o_2)\) of \( Q \times D \), such that stratum \( k - 1 \) has an edge that enters node \((v_1, o_1)\). The size of the stratum traversal \( T \) is \( O(q^2d) \), where \( q \) is the size of the query \( Q \) and \( d \) is the size of the database \( D \). The stratum traversal \( T \) can be constructed in \( O(q^2d) \) time.

Let \( T \) be the stratum traversal that was constructed as described above. The maximal OR-matchings are generated by an incremental algorithm that iterates over the edges in \( T \). Initially, the set of current subgraphs of \( Q \times D \) has only one member,
Chapter 4  Queries with Maximal Answers  95

namely, the root of $Q \times D$. In the $i$th iteration, each current subgraph is expanded, if possible, by adding the edge that appears in position $i$ of $T$ (the nodes of the edge are also added, unless they are already in the subgraph).

The Various Cases of Adding An Edge

Let $G$ be a subgraph of $Q \times D$, such that $G$ satisfies Conditions 1–3 of Proposition 4.8; that is,

- $G$ has no repeated variables;
- $G$ contains the root of $Q \times D$; and
- All the nodes of $G$ are reachable from the root (via edges that belong to $G$).

Consider an attempt to add an edge $(v_1, o_1)(v_2, o_2)$ of $Q \times D$ to $G$. Several cases are possible and we now consider them one by one.

Case 1 The node $(v_2, o_2)$ satisfies $v_2 = r_Q$ and $o_2 \neq r_D$. In this case, the edge $(v_1, o_1)(v_2, o_2)$ is not added to $G$, since the node $(v_2, o_2)$ cannot belong to a subgraph of $Q \times D$ that satisfies Conditions 1–3 of Proposition 4.8. Thus, $G$ remains unchanged.

Case 2 The node $(v_1, o_1)$ is not in $G$. In this case, the edge $(v_1, o_1)(v_2, o_2)$ cannot be added to $G$ and $G$ remains unchanged.

Case 3 The nodes $(v_1, o_1)$ and $(v_2, o_2)$ as well as the edge that connects them are all in $G$. In this case, $G$ remains unchanged.

Case 4 Both $(v_1, o_1)$ and $(v_2, o_2)$ are in $G$, but the edge that connects them is not in $G$. In this case, the edge $(v_1, o_1)(v_2, o_2)$ is added to $G$. Notice that the result is a subgraph $G'$ of $Q \times D$ that satisfies Conditions 1–3 of Proposition 4.8. The new graph $G'$ replaces $G$.

Case 5 The node $(v_1, o_1)$ is in $G$ and $v_2$ does not appear in any node of $G$. In this case, the edge $(v_1, o_1)(v_2, o_2)$ as well as the node $(v_2, o_2)$ are added to $G$. Notice that the result is a subgraph $G'$ of $Q \times D$ that satisfies Conditions 1–3 of Proposition 4.8. The new graph $G'$ replaces $G$. 
Case 6 The node \(v_1, o_1\) is in \(G\), \(v_2 \neq r_Q\) and \(G\) also has a node \((v_2, o_3)\), where \(o_2 \neq o_3\). In this case, the addition of the edge \((v_1, o_1)l(v_2, o_2)\) to \(G\) forces the deletion of the node \((v_2, o_3)\). Thus, a new subgraph \(G'\) is created from \(G\) as follows. First, the node \((v_2, o_2)\) and the edge \((v_1, o_1)l(v_2, o_2)\) are added to \(G\). Second, the node \((v_2, o_3)\) is deleted from \(G'\). Third, all the nodes that are not reachable from the root, due to the deletion of \((v_2, o_3)\), are deleted from \(G'\); edges that are incident on deleted nodes are also deleted. The result is a subgraph \(G'\) of \(Q \times D\) that satisfies Conditions 1–3 of Proposition 4.8. Notice that \(G\) is not subsumed by \(G'\), since the former includes the node \((v_2, o_3)\), but the latter does not. However, it could be that \(G\) subsumes \(G'\) (that may happen if the edge \((v_1, o_1)l(v_2, o_2)\) that was initially added to \(G\) is eventually deleted). In summary, both \(G\) and \(G'\) should be retained unless \(G'\) is subsumed by \(G\).

The Algorithm

The algorithm computes a set \(\mathcal{M}\) as follows.

1. Initially, \(\mathcal{M}\) has one graph that consists of the root of \(Q \times D\).

2. Repeat the next two steps for each edge \(e\) in the stratum traversal \(T\), starting with the first edge of \(T\).

3. For each graph \(G \in \mathcal{M}\), replace \(G\) with the result of adding \(e\) to \(G\), according to the above six cases. Recall that \(G\) is replaced either with \(G\) itself, with a new graph \(G'\) that subsumes \(G\) or with \(G\) and a new graph \(G'\), such that neither one subsumes the other.

4. Remove subsumed graphs from \(\mathcal{M}\), i.e., a graph \(G \in \mathcal{M}\) is removed if it is subsumed by some other graph of \(\mathcal{M}\) (and this step is repeated until there are no subsumptions among the graphs of \(\mathcal{M}\)).

The next theorem shows that the algorithm is correct and gives its time complexity.
Theorem 4.9 (Correctness) Given a query $Q$ and a database $D$, the algorithm terminates with $\mathcal{M} = \mathcal{M}_{or}(Q, D)$. The running time of the algorithm is $O(q^3dm^2)$, where $q$ is the size of $Q$, $d$ is the size of $D$ and $m$ is the size of $\mathcal{M}_{or}(Q, D)$.

Correctness of the Algorithm

The algorithm iterates over all the edges in the stratum traversal $T$. In each iteration, Steps 3 and 4 are executed once. The $i$th iteration is for the edge that occurs in position $i$ of $T$. The value of $\mathcal{M}$ at the end of the $i$th iteration is denoted as $\mathcal{M}^i$.

Consider a maximal or-subgraph $M$ of $Q \times D$. Intuitively, we would like to prove that for all $i$, there is a graph $G \in \mathcal{M}^i$, such that $G$ subsumes the restriction of $M$ just to the edges that appear in the first $i$ positions of $T$. However, the minimal depth of an edge in $M$ may be greater than its minimal depth in $Q \times D$. Therefore, a more elaborate definition is needed.

Consider a position $i$ in the stratum traversal $T$ and suppose that position $i$ occurs in the stratum that has depth $d$. Let $M$ be a maximal or-matching subgraph of $Q \times D$. The $i$-portion of $M$, denoted $M^i$, consists of all the edges $m_1m_2$ of $M$, such that the minimal depth of $m_1m_2$ in $M$ is not greater than $d$ and there is at least one occurrence of $m_1m_2$ among the first $i$ positions of $T$. Note that, by definition, the 0-portion of $M$ consists just of the root of $M$. Also note that $M^i$ is an or-matching subgraph of $Q \times D$.

Next, we will prove the following lemma that shows the correctness of the algorithm.

Lemma 4.10 When the algorithm completes its execution, $\mathcal{M} = \mathcal{M}_{or}(Q, D)$.

Proof. The proof follows from the following two claims.

Claim 4.11 During the evaluation of the algorithm, all the graphs in $\mathcal{M}$ are or-matchings graphs.

This claim is true, since all the graphs in $\mathcal{M}$ satisfy Conditions 1–3 of Proposition 4.8 throughout the evaluation of the algorithm.
Claim 4.12 When the algorithm terminates, the following holds. For every maximal or-matching graph $M$, the set $\mathcal{M}$ has a graph $G$, such that $G$ subsumes $M$.

Together, the two claims imply that at the end of the algorithm, $\mathcal{M}$ is exactly the set of all maximal or-matching graphs, since the set $\mathcal{M}$ is subsumption free.

Claim 4.12 is proved by induction, using the following inductive hypothesis: If $M$ is a maximal or-graph, then $M^i$ is subsumed by some $G \in \mathcal{M}^i$.

For the basis of the induction, we should consider the initialization of $\mathcal{M}$ just prior to the first iteration. Initially, $\mathcal{M}$ contains one subgraph $G$ of $Q \times D$ that comprises the root of $Q \times D$. The 0-portion of $M$ is just the root, so the induction hypothesis is true. Now suppose that the inductive hypothesis holds at the end of the $i$th iteration. We will show that it also holds at the end of the $(i + 1)$st iteration.

Let $M$ be a maximal or-matching graph and $(v_1, o_1)l(v_2, o_2)$ be the $(i + 1)$st edge in the stratum traversal $T$. We need to show that the induction hypothesis holds after expanding the graphs in $\mathcal{M}^i$, by adding $(v_1, o_1)l(v_2, o_2)$.

We consider two cases, depending on whether $(v_1, o_1)l(v_2, o_2)$ is an edge of $M^{i+1}$ or not. We show that in both cases in $\mathcal{M}^{i+1}$ there is a $G'$ that subsumes $G$.

Suppose that $(v_1, o_1)l(v_2, o_2)$ is not an edge of $M^{i+1}$. Then, $M^i$ and $M^{i+1}$ are identical. By the inductive hypothesis, there is a graph $G$ in $\mathcal{M}^i$, such that $M^i$ is subsumed by $G$. Since $G \in \mathcal{M}^i$, there is a $G' \in \mathcal{M}^{i+1}$, such that $G$ is subsumed by $G'$. Thus, $M^{i+1}$ is subsumed by $G'$ and the inductive hypothesis is proven.

Now, suppose that $(v_1, o_1)l(v_2, o_2)$ is an edge of $M^{i+1}$. By the inductive hypothesis, there is a graph $G$ in $\mathcal{M}^i$, such that $M^i$ is subsumed by $G$.

An important observation is that $(v_1, o_1)$ is a node of $G$. This observation follows from the following assertions. First, $M^{i+1}$ is an or-matching graph and $(v_1, o_1)$ is reachable from the root by a path that comprises edges of $M^i$. Thus, $(v_1, o_1)$ is a node of $M^i$. Second, since $G$ subsumes $M^i$, $(v_1, o_1)$ is a node of $G$.

Next, we consider the six cases of adding the edge $(v_1, o_1)l(v_2, o_2)$ to $G$ during the $(i + 1)$st iteration, as described in Section 4.2.2. We show that in all the cases, $\mathcal{M}^{i+1}$ contains a graph that subsumes $M^{i+1}$.

The first two cases contradict the assumption that $(v_1, o_1)l(v_2, o_2)$ is an edge of
$M^{i+1}$. In the first case, the edge $(v_1, o_1)l(v_2, o_2)$ cannot be in $M$ and, thus, it is not an edge of $M^{i+1}$. In the second case, the node $(v_1, o_1)$ is not in $G$, in contradiction to the above observation.

In the third case, $G$ already has the edge $(v_1, o_1)l(v_2, o_2)$. Since $G$ subsumes $M^i$, it follows that it also subsumes $M^{i+1}$. Therefore, since $G'$ subsumes $G$ it also subsumes $M^{i+1}$.

The fourth and the fifth cases are similar. The graph $G'$ has all the nodes and edges of $G$ and also the added edge $(v_1, o_1)l(v_2, o_2)$. Since $G$ subsumes $M^i$, $G'$ subsumes $M^{i+1}$.

Finally, consider the sixth case. First note that $M$ cannot include the node $(v_2, o_3)$. This is because $M^{i+1}$ includes the node $(v_2, o_2)$ and, hence, $M$ includes $(v_2, o_2)$. $M$ is an assignment graph so it cannot have two nodes with the same variable.

In the sixth case, a new subgraph $G'$ is created from $G$ by adding the edge $(v_1, o_1)l(v_2, o_2)$ and then deleting the node $(v_2, o_3)$ as well as all the nodes that are not reachable from the root. Let $\tilde{G}$ be the subgraph that is obtained immediately after the addition of $(v_1, o_1)l(v_2, o_2)$ and just before the deletion of any node. Since $M$ does not include $(v_2, o_3)$, all the nodes of $M^{i+1}$ are reachable from the root via paths that do not include $(v_2, o_3)$. By the induction hypothesis, $M^i$ is subsumed by $G$ and, consequently, all the nodes and edges of $M^{i+1}$ also appear in $\tilde{G}$. In $\tilde{G}$, all the nodes of $M^{i+1}$ are reachable from the root via paths that do not include $(v_2, o_3)$. Thus, none of the nodes and none of the edges of $M^{i+1}$ is deleted from $\tilde{G}$. Therefore, $G'$ subsumes the graph $M^{i+1}$.

In summary, in all of the above cases, the inductive hypothesis holds at the end of the $(i + 1)$st iteration.

\[\square\]

**Time Complexity**

In this section, we will prove the part of Theorem 4.9 that refers to the time complexity. The proof hinges on the fact that the number of graphs in $\mathcal{M}$ does not decrease from one iteration to the next, as shown by the following lemma.
Lemma 4.13 For each \( i \geq 0 \), \(|\mathcal{M}^i| \leq |\mathcal{M}^{i+1}|\).

Proof. The lemma is proved by showing that each \( G' \in \mathcal{M}^{i+1} \) subsumes at most one \( G \in \mathcal{M}^i \). To derive a contradiction, we assume that \( G_1 \) and \( G_2 \) are two graphs in \( \mathcal{M}^i \) that are subsumed by some \( G' \in \mathcal{M}^{i+1} \). Suppose that position \( i \) of the stratum traversal \( T \) occurs in a stratum that has a depth \( d \). Consider the graph \( \hat{G} \) that consists of all the edges that appear in either \( G_1 \) or \( G_2 \), as well as all the nodes connected by those edges. (If there are no edges, then \( \hat{G} \) consists just of the root of \( Q \times D \).) Clearly, each edge of \( \hat{G} \) has a depth (in \( \hat{G} \)) that is not greater than \( d \) and has an occurrence among the first \( i \) positions of the stratum traversal \( T \). Furthermore, \( \hat{G} \) has no repeated variables, since it is subsumed by another or-matching subgraph, \( G' \). Thus, \( \hat{G} \) is an or-matching subgraph of \( Q \times D \), and \( \hat{G}^i \) (the \( i \)-portion of \( \hat{G} \)) is the same as \( \hat{G} \). By the inductive hypothesis in the proof of Lemma 4.10, there is a \( G \in \mathcal{M}^i \), such that \( G \) subsumes \( \hat{G} \). Consequently, \( G \) subsumes both \( G_1 \) and \( G_2 \), in contradiction to the fact that \( \mathcal{M}^i \) is subsumption free.

We will now complete the proof of the time complexity. Recall that \( q \), \( d \) and \( m \) are the sizes of the query, the database and the result, respectively. By Lemma 4.13, at any time during the evaluation of the algorithm, before the removal of subsumed graphs, \( \mathcal{M} \) has at most twice as many graphs as the final value of \( \mathcal{M} \). After the removal of subsumed graphs, the number of graphs in \( \mathcal{M} \) does not exceed the number of graphs in the result. In Step 3 of the algorithm, adding an edge to a graph \( G \in \mathcal{M} \) takes \( O(q) \) time. Thus, each execution of Step 3 takes \( O(qm) \) time. Each execution of Step 4 takes \( O(qm^2) \) time. The size of the stratum traversal is \( O(q^2d) \) and it can be constructed in \( O(q^2d) \) time. Hence, Steps 3 and 4 are repeated \( O(q^2d) \) times and the whole algorithm takes \( O(q^3dm^2) \) time.

4.2.3 Computing Maximal Weak Matchings

To modify the algorithm so that it will compute the set of all maximal weak matchings, Cases 5 and 6 of Section 4.2.2 should be changed. In each of these two cases, after creating the subgraph \( G' \), there is a need to test and possibly modify \( G' \) so that it will preserve the edges of \( Q \). This is done as follows.
Let \((v_1, o_1)l(v_2, o_2)\) be the edge that was added to \(G'\). For each node \((v_i, o_i)\) in \(G'\), if there is a label \(h\), such that \(v_i h v_2\) is an edge of \(Q\), but \(o_i h o_2\) is not an edge of \(D\), then delete node \((v_i, o_i)\) from \(G'\). Similarly, for each node \((v_j, o_j)\) in \(G'\), if there is a label \(h\), such that \(v_2 h v_j\) is an edge of \(Q\), but \(o_2 h o_j\) is not an edge of \(D\), then delete node \((v_j, o_j)\) from \(G'\). Nodes that become non-reachable from the root should also be deleted. Edges that are incident on deleted nodes are deleted as well. As a result of these deletions, \(G'\) may no longer subsume \(G\) and consequently, in Case 5, both \(G'\) and \(G\) should be added to \(\mathcal{M}\) (or just \(G\) should be added if it subsumes \(G'\)). The same is done in Case 6.

**Theorem 4.14** Given a query \(Q\) and a database \(D\), the set \(\mathcal{M}_w(Q, D)\) of all maximal weak matchings can be computed in \(O(q^3 dm^2)\) time, where \(q\) is the size of \(Q\), \(d\) is the size of \(D\) and \(m\) is the size of \(\mathcal{M}_w(Q, D)\).

In proving the correctness of the algorithm for computing the maximal weak matchings, Claim 4.11 should be replaced with the following claim.

**Claim 4.15** During the evaluation of the algorithm, the closure of each graph in \(\mathcal{M}\) is a weak matching.

Claim 4.12 remains the same, except for changing “maximal OR-matching graph \(M\)” to “maximal weak-matching graph \(M\).” The proofs of correctness and time complexity are similar to the ones in the case of computing the maximal OR-matchings.

### 4.3 Summary of Contributions

Chapter 4 follows the work of [41, 42] on queries with incomplete answers over semistructured data. In [41, 42], three semantics, AND, OR and weak, were presented and the complexity of query evaluation was investigated. In particular, it was shown that for DAG queries, both the OR-semantics and the weak semantics have polynomial-time evaluation algorithms under input-output complexity. In this chapter, we have shown that this result carries over to cyclic queries.
Chapter 5

Combining Flexibility with Maximal Answers

In the previous chapters we introduced two paradigms: The paradigm of flexible queries and the paradigm of maximal answers. In the current chapter, we show how to combine the two paradigms. Combining the two paradigms facilitates querying of databases that are both irregular and incomplete.

The outline of this chapter is as follows. In Section 5.1, four new semantics are defined, namely, the semiflexible-or, the semiflexible-weak, the flexible-or and the flexible-weak semantics. In Section 5.2, query evaluation under the four new semantics is investigated. Finally, in Section 5.3 we summarize the contribution of this chapter.

5.1 Queries under the Combined Semantics

In this section we define new semantics that are called combined semantics. The combined semantics generalize the weak and the or-semantics and add flexibility to matchings, according to the paradigm of flexible queries.

5.1.1 Combined Semantics

We start by providing required definitions. We also provide a characterizations for weak and or-matchings. We will use this characterization when we define the combined semantics.
Chapter 5  Combining Flexibility with Maximal Answers

Definition 5.1 (Rooted Fragment) Given a query \( Q = (V, E_Q, r_Q) \), a rooted fragment of \( Q \) is a rooted graph \( F = (V_F, E_F, r_Q) \), where \( V_F \) is a subset of \( V \) and \( E_F \) is a subset of \( E_Q \cap (V_F \times V_F) \). The rooted fragment \( F \) preserves the edges of \( Q \) if each edge of \( Q \) that lies on two nodes of \( F \) is contained in \( E_F \), that is, all the edges of \( E_Q \) that connect nodes of \( V_F \) belong to \( E_F \). \(^1\)

The next proposition characterizes weak and OR-matchings in terms of rooted fragments.

Proposition 5.2 (Weak-Matchings and OR-Matchings) Let \( Q \) be a query and \( D \) be a database. A mapping \( \mu \) is an OR-matching of \( Q \) w.r.t. \( D \) if and only if there is a rooted fragment \( F \) of \( Q \), such that \( \mu \) is a rigid mapping of \( F \) w.r.t. \( D \) and \( \mu \) maps to null all the variables of \( Q \) that are not in \( F \). The mapping \( \mu \) is a weak-matching of \( Q \) w.r.t. \( D \) if and only if the rooted fragment also preserves the edges of \( Q \).

Proof. Suppose that \( F \) is a rooted fragment of \( Q \) and \( \mu \) is a rigid mapping of \( F \) w.r.t. \( D \). We show that \( \mu \) is an OR-matching of \( Q \) w.r.t. \( D \).

First, we create an assignment graph \( G \). The nodes of \( G \) are pairs of the form \((v, \mu(v))\), where \( v \) is a variable of \( F \). For each edge \( uv \) if \( F \) there is an edge \((u, \mu(u))(v, \mu(v))\) in \( G \). To see why \( G \) is an assignment graph, one should observe that if \( uv \) is an edge in \( F \) then there is an edge \( \mu(v)\mu(u) \) in \( D \). This is because \( \mu \) satisfies all the ec’s of \( F \).

Since \( \mu \) maps the query root \( r_Q \) to database root \( r_D \), the pair \((r_Q, r_D)\) is a node of \( G \). Because in \( F \) each node is reachable from the root, also in \( G \) each node is reachable from the root. Thus, according to Definition 4.3, \( \mu \) is an OR-matching.

The correctness of the other direction follows from the following observation. If \( M \) is an OR-matching graph of \( \mu \), then the nodes and edges of \( Q \) that appear in nodes and edges of \( M \) form a rooted fragment of \( Q \). For weak matchings, the arguments are similar.

Essentially, according to Proposition 5.2 an assignment \( \mu \) is an OR-matching (weak-matching) if there exists a rooted fragment (a rooted fragment that preserves

\(^1\)Note that in literature, sometimes a fragment \( F \) that preserves the edges of \( Q \) is called “the induced subgraph of \( Q \) on the nodes of \( V_F \)”.
the edges) of Q such that μ satisfies the rc and all the ≢c’s of the fragment. In order to add flexibility to the weak and or-semantics, we replace the query constraints in the definitions as defined next.

**Definition 5.3 (Combined Semantics)** Suppose that Q is a query and D is a database. Let ξ be one of the three semantics: rigid, semiflexible and flexible. Consider the following four conditions.

1. F is a rooted fragment of Q.
2. μ is a ξ-matching of F w.r.t. D.
3. μ maps to null all the variables of Q that are not in F.
4. F preserves the edges of Q.

A mapping μ is a ξ-or-matching of Q w.r.t. D if the first three conditions hold. It is a ξ-weak-matching of Q w.r.t. D if all four conditions are true.

For tree queries there are actually only three combined semantics, as pointed out by the following proposition.

**Proposition 5.4 (Tree Queries)** Suppose that Q is a tree query, D is a database and ξ is one of the three semantics: rigid, semiflexible and flexible. An assignment μ is a ξ-or-matching of Q w.r.t. D if and only if μ is a ξ-weak-matching of Q w.r.t. D.

**Proof.** In a tree, every rooted fragment preserves the edges.

For queries that are arbitrary graphs, the following holds.

**Proposition 5.5 (Weak and OR-Matchings)** Suppose that Q is a query, D is a database and ξ is one of the three semantics: rigid, semiflexible and flexible. If μ is a ξ-weak-matching of Q w.r.t. D then μ is a ξ-or-matching of Q w.r.t. D.

**Proof.** If μ satisfies all the four conditions of Definition 5.3 then it satisfies the first three conditions of the definition.
Table 5.1: The flexible-weak matchings of Query 1, Query 2 and Query 3 of Figure 2.2 w.r.t. the movie database of Figure 2.1. Only matchings in which both \( x \) and \( u \) are not null are presented.

Query results, under the combined semantics, consist only of maximal answers. If \( Q \) is a query and \( D \) is a database, the set of maximal flexible-or-matchings of \( Q \) w.r.t. \( D \) is denoted as \( \mathcal{M}_{f-or}(Q, D) \). The set of maximal flexible-weak-matchings of \( Q \) w.r.t. \( D \) is denoted as \( \mathcal{M}_{f-w}(Q, D) \). The notations for the combined semantics in which the flexible semantics is replaced with the semiflexible semantics are similar (i.e., \( \mathcal{M}_{sf-or}(Q, D) \) and \( \mathcal{M}_{sf-w}(Q, D) \)).

5.1.2 Examples

Example 5.6 Recall the three queries that are depicted in Figure 2.2. All three queries look for actor-movie pairs in which the actor acted in the movie. For actors, the queries return their name. For movies, the queries return their title and the year of production.

We examine the case where the three queries are posed, under the flexible-weak semantics, to the movie database that is shown in Figure 2.1. Table 5.1 presents the flexible-weak matchings in which \( x \) and \( u \) are not assigned a null value, i.e., matchings that consist of both a movie and an actor. Note that with the requirement that \( x \) and \( u \) should not be null, all three queries have the same set of matchings. For Query 1 and Query 3, there is an additional flexible-weak matching where \( r \), \( x \) and \( y \) are mapped to 1, 13 and 33, respectively, and all the other variables are mapped to null. This matching is not shown in Table 5.1 because it maps \( u \) to null. In the table, for atomic objects, their value is presented in addition to the object identifier.

Consider a user that looks for related actor-movie pairs but is familiar only with
Table 5.2: The semiflexible-weak matchings of Query 1, Query 2 and Query 3 of Figure 2.2 w.r.t. the movie database of Figure 2.1.

the ontology of the database and not with the structure of the database. This user may formulate one of the three queries of Figure 2.2. Under the flexible-weak semantics, all three queries will return a correct answer, that is, all the actor-movie pairs in which the actor acted in the movie. As shows in Chapter 1, this is not the case when using the traditional rigid semantics. The example also demonstrates how the flexible-weak semantics facilitates the querying of data which is both irregular and incomplete.

Under the flexible-or semantics, Query 1 and Query 2 will return the same set of matchings as under the flexible-weak semantics. This is because the queries are trees. Query 3, however, may return, under the flexible-or semantics, actors and movies that are not related. The reason for this is that under the flexible-or semantics, it is not required that matchings will preserve the edges of the query and, in particular, the edge that goes from \( u \) to \( x \).
**Example 5.7** Table 5.2 shows the semiflexible-weak matchings of the three queries of Figure 2.2 w.r.t. the movie database. The semiflexible semantics is more restrictive than the flexible semantics. Therefore, for each semiflexible-weak matching in Table 5.2 there is a flexible-weak matching, of the same query, that subsumes it. This shows that queries under the semiflexible-weak semantics may extract less information than under the flexible-weak semantics. Yet, the semiflexible-weak semantics reduces the chance of having non-related objects in a matching, in comparison to the flexible-weak semantics. Most importantly, under the semiflexible-weak semantics, all three queries find all the related actor-movie pairs. Furthermore, actor-movie pairs that are not related are never returned.

### 5.2 Computing Maximal Matchings

In this section, we study the problem of computing maximal matchings under the combined semantics that were presented above. We describe evaluation algorithms for queries with combined semantics and analyze the input-output complexity of these algorithms.

#### 5.2.1 Computing Semiflexible-or-Matchings

Evaluation of maximal semiflexible-or-matchings is discussed next. At the beginning of this section, we show that evaluation of dag queries is NP-hard. Then we provide a polynomial-time algorithm for the case where the query is either a tree or a dag in which the number of nodes, with more than one parent, is bounded. At the end of the section, we provide an algorithm for computing maximal semiflexible-or-matchings of cyclic queries.

**DAG Queries**

Consider the evaluation of dag queries under the semiflexible-or semantics. We show that it is not likely that this evaluation has an algorithm with polynomial-time input-output complexity. For a proof we use a reduction of 3SAT.
Lemma 5.8 (Reduction of 3SAT) For a given 3CNF formula $\varphi$ over a set of propositional letters $P$, one can construct in polynomial time a dag query $Q$ and a dag database $D$, such that the following are equivalent.

- The set of maximal semiflexible-or-matchings consists of a single matching and this matching does not map any variable of $Q$ to null.
- There is an assignment, for the propositional letters in $P$, that satisfies $\varphi$.

The proof of Lemma 5.8 is given in Appendix C.

Theorem 5.9 (Complexity of semiflexible-or-queries) If there exists an algorithm that computes $\mathcal{M}_{sf-or}(Q, D)$ in time polynomial in the size of the input ($Q$ and $D$) and the output, then $\text{PTIME} = \text{NP}$.

Proof. Let $q$, $d$ and $m$ be the sizes of the query, the database and the result, respectively. Suppose that there is an evaluation algorithm $A$ and there is a polynomial $P(x_1, x_2, x_3)$ such that for every database $D$ and query $Q$, the algorithm $A$ computes $\mathcal{M}_{sf-or}(Q, D)$ in $O(P(q, d, m))$ time. We show that this would yield a polynomial algorithm to decide 3SAT. However, 3SAT is NP-complete.

Let $\varphi$ be a 3CNF formula over a set $P$ of propositional letters. Let $D_\varphi$ and $Q_\varphi$ be the query and the database that are constructed according to Lemma 5.8. Consider a satisfying assignment $f$ to the propositional letters of $P$. If $f$ satisfies $\varphi$, then $\mathcal{M}_{sf-or}(Q_\varphi, D_\varphi)$ contains a single element. Furthermore, the evaluation of $\mathcal{M}_{sf-or}(Q_\varphi, D_\varphi)$ is in $O(P(|Q_\varphi|, |D_\varphi|, 1))$ time. Since the sizes of $D_\varphi$ and $Q_\varphi$ are polynomial in the size of $\varphi$, there exists a polynomial $P'(x)$ such that $A$ terminates after time $P'(|\varphi|)$, i.e., after polynomial time in the size of $\varphi$.

In order to solve 3SAT in polynomial time, we would construct $D_\varphi$ and $Q_\varphi$ and then execute $A$ with $D_\varphi$ and $Q_\varphi$ as the input. If $A$ does not terminate after $P'(|\varphi|)$ time, we stop the run and know that $\varphi$ is unsatisfiable. If $A$ stops without our intervention, we examine the result. If the result consists of a single matching and this matching maps all the variables of $Q$ to database objects (i.e., not to null), then we know that $\varphi$ is satisfiable. Otherwise, we know that $\varphi$ does not have a satisfying assignment. \[\boxed{}\]
Chapter 5 Combining Flexibility with Maximal Answers

The above theorem shows that there is no polynomial-time algorithm for evaluating dag queries under the semiflexible-or-semantics, assuming that P ≠ NP. However, consider a query Q, a database D and a mapping μ that assigns objects of D to the variables of Q. To decide if μ is a semiflexible-or-matching of Q w.r.t. D is in NP. This is because one can guess a rooted fragment F of Q and, according to Theorem 3.19, verify in polynomial time, in the size of Q and D, whether μ is a semiflexible matching of Q w.r.t. F.

Next, we provide an algorithm that computes maximal semiflexible-or matchings. First we describe the algorithm for dag queries and then modify this algorithm, to deal with cyclic queries. In the general case, the input-output complexity of the algorithm is not polynomial. Yet, there are cases in which the algorithm is tractable. In particular, for tree queries, the algorithm has a polynomial-time input-output complexity. An analysis of the algorithm runtime will be given at the end of the section.

Before presenting the algorithm, we show that when computing semiflexible-or-matchings, it is sufficient to compute the matchings w.r.t. subtrees of the query instead of examining all the rooted fragments of the query, including fragments that are not trees.

**Proposition 5.10 (Tree Expansion)** Consider a query Q and a database D. An assignment μ is a semiflexible-or-matching of Q w.r.t. D if and only if there is a tree T that satisfies the following three conditions.

1. T is a rooted fragment of Q;
2. μ is a semiflexible-matching of T w.r.t. D; and
3. μ maps to null all the variables of Q that are not in T.

**Proof.** Obviously, if there exists a tree T that satisfies the conditions of the proposition, then μ is a semiflexible-or-matching. This follows directly from Definition 5.3.

We show the other direction. Suppose that μ is a semiflexible-or-matching. According to Definition 5.3, there is rooted fragment F of Q such that μ is a
semiflexible-matching of $F$ w.r.t. $D$ and $\mu$ maps to null all the variables of $Q$ that are not in $F$. If $F$ is a tree, then the claim is correct and there is nothing to prove. Otherwise, let $T$ be a spanning tree of $F$, i.e., a tree that consists of all the nodes of $F$. Obviously, $T$ is a rooted tree and $\mu$ maps to null all the variables of $Q$ that are not in $T$. Furthermore, $\mu$ is a semiflexible matching of $F$ w.r.t. $D$. Thus, $\mu$ satisfies the SF-Condition w.r.t. every path of $F$. Since all the paths of $T$ are also paths of $F$, $\mu$ satisfies the SF-Condition w.r.t. every path of $T$. Therefore, $\mu$ is a semiflexible matching of $T$ w.r.t. $D$. 

Next, we present an evaluation algorithm for queries under the semiflexible-or-semantics. Generally, the evaluation algorithm computes rooted subtrees of the query and finds the semiflexible matchings w.r.t. these trees. For trees and for dag queries, the evaluation is performed by a series of extension steps. We describe extension steps, first, and then describe how to apply these extension steps when evaluating a query.

**Definition 5.11 (Extension Set)** Let $T = (V_T, E_T, r_Q)$ be a rooted subtree of a query $Q$ and let $\mu$ be a semiflexible matching of $T$ w.r.t. a database $D$. Consider a variable $v$ of $Q$ that is not in $V_T$. The set $\text{Ext}_{sf-or}(T, \mu, v)$ is the set of all pairs $(T', \mu')$ such that:

1. $T'$ is a rooted subtree of $Q$.
2. The variables of $T'$ are $V_T \cup \{v\}$ and the edges of $T'$ consists of $E_T$ plus one edge that connects a node in $V_T$ to $v$.
3. The matchings $\mu$ and $\mu'$ are equal on all the nodes of $T$.
4. The matching $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$.

Intuitively, $\text{Ext}_{sf-or}(T, \mu, v)$ contains pairs of a tree $T'$ and a matching $\mu'$, where $T'$ is an extension of $T$ ($T'$ is created by adding to $T$ an edge that connects $T$ with $v$) and $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$.

Computing $\text{Ext}_{sf-or}(T, \mu, v)$ is described next. First, we find all the edges of $Q$ that connect a node of $T$ to $v$. Adding each of these edges to $T$ provides a tree $T'$
that extends $T$ to include $v$. For an object $o$ in $D$, let $\mu \cup \{(v,o)\}$ be a matching that is equal to $\mu$ on the variables of $T$ and assigns $o$ to $v$. Consider a pair of a tree $T'$ that is the result of extending $T$ with $v$ and a matching $\mu' = \mu \cup \{(v,o)\}$. If $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$, we add the pair $(T',\mu')$ to the result. Checking if $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$ is being done by testing the two conditions that are described in the next proposition.

**Proposition 5.12 (Testing Extended Matchings)** Let $\mu'$ be a matching that is created by extending $\mu$ with an assignment of $o$ to $v$, as described above. Then $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$ if the next two conditions are satisfied.

1. Let $l$ be the label on the edge that enters $v$ in $T'$. There is an edge in $D$ that enters $o$ and is labeled with $l$.

2. If $v_a$ is an ancestor of $v$ in $T'$ and has an incoming label $l_a$, then $D$ either has a path $\mu(v_a)\ast l o$ or a path $o\ast l_a\mu(v_a)$.

**Proof.** First, we show why it is sufficient to check the conditions in the proposition only w.r.t. $v$ and $o$ and not w.r.t. all the variables of the query. Recall that the matching $\mu$ is a semiflexible matching of $T$ w.r.t. $D$. Thus, every path of $T$ satisfies the SF-Condition w.r.t. $\mu$. Every path in $T'$ that does not include $v$ is a path in $T$ and, hence, satisfies the SF-condition w.r.t. $\mu'$. So, in order to check that $\mu'$ is a semiflexible matching of $T'$ w.r.t. $D$, we merely need to check that paths that contain $v$ satisfy the SF-condition. This is precisely what the conditions of the proposition do. Therefore, if the two conditions of the proposition are satisfied, then $\mu'$ is a connectivity-preserving mapping. Note that the first condition of the proposition is sufficient and necessary for satisfaction of the first condition in the definition of a connectivity-preserving mapping (Definition 3.10). According to Corollary 3.13, $\mu'$ is a semiflexible matching of $T'$.

Consider two pairs $(T,\mu)$ and $(T',\mu')$, where (1) $T$ and $T'$ are subtrees of $Q$; and (2) $\mu$ and $\mu'$ are semiflexible matchings w.r.t. $T$ and $T'$, respectively. We say that $(T',\mu')$ **subsumes** the pair $(T,\mu)$ if (1) all the edges and nodes of $T$ are also edges
and nodes of $T'$, and (2) for each variable $v$ of $T$, $\mu(v) = \mu'(v)$. During the run of the algorithm, subsumed pairs are discarded in order to prevent redundant work.

Let $Q$ be a dag query and $D$ be a database. Next, we present the algorithm $SFORD$ that computes the set of maximal semiflexible-or matchings of $Q$ w.r.t. $D$.

$SFORD(Q,D)$

1. Sort topologically the nodes of $Q$. Let $v_0, \ldots, v_n$ be the nodes of $Q$ according to the topological order.

2. Create an initial set $M^0$ that contains a single pair $(T,\mu)$, where $T$ has no edges and has a single node—the query root. The matching $\mu$ is a mapping of the query root to the database root.

3. Create the sets $M^1, \ldots, M^n$, iteratively. For each $1 \leq i \leq n$, the set $M^i$ is created by extending the matchings of $M^{i-1}$ w.r.t. $v_i$. Extension step $i$ is performed as follows.

   (a) Start with an empty set $M^i$.

   (b) For each pair $(T,\mu)$ in $M^{i-1}$, if $Ext_{sf-or}(T,\mu, v_i)$ is empty, add to $M^i$ the pair $(T,\mu)$. Else, add to $M_i$ all the elements of $Ext_{sf-or}(T,\mu, v_i)$.

   (c) Iteratively, remove an element of $M^i$ that is subsumed by another element of $M^i$, until there are no subsumptions.

4. Let $M$ be the set $\{\mu \mid (T,\mu) \in M^n\}$. That is, $M$ consists of all the matchings $\mu$ that are part of a pair $(T,\mu)$ in $M^n$.

5. For each matching $\mu$ in $M$, if there are variables of $Q$ that $\mu$ does not map to any object, then replace $\mu$ by a mapping $\tilde{\mu}$, such that $\tilde{\mu}$ is equals to $\mu$ on all the variables that $\mu$ maps to database objects. In addition, $\tilde{\mu}$ maps to null variables for which $\mu$ is not defined.

6. Remove from $M$ subsumed matchings, i.e., non-maximal elements, and return $M$. 
The next propositions prove the correctness of the algorithm and analyze the input-output time complexity.

**Proposition 5.13 (Completeness)** Suppose that $Q$ is a dag query and $D$ is a database. If $\mu$ is a semiflexible-or-matching of $Q$ w.r.t. $D$, then the result of Algorithm SFORD contains $\mu$ or contains a matching that subsumes $\mu$.

**Proof.** According to Proposition 5.10, there exists a rooted subtree $T$ of $Q$, such that $\mu$ is a semiflexible matching of $T$ w.r.t. $D$. We use induction to show that either $\mathcal{M}^n$ contains $(T, \mu)$ or $\mathcal{M}^n$ contains a pair $(T', \mu')$ that subsumes $(T, \mu)$.

Let $v_0, \ldots, v_i$ be the first $i+1$ variables in the topological order over the variables of $Q$. We denote by $T_i$ the restriction of $T$ to $v_0, \ldots, v_i$. The variables of $T_i$ are the elements of the intersection $V_i = \{v_0, \ldots, v_i\} \cap V_T$, where $V_T$ are the variables of $T$. The edges of $T_i$ are all the edges of $T$ that connect two variables of $V_i$. We denote by $\mu_i$ the restriction of $\mu$ to $v_0, \ldots, v_i$. That is, $\mu(v) = \mu_i(v)$ for each $v$ in $V_i$.

The induction hypothesis is the following. In $\mathcal{M}^i$ there is a pair $(T_i', \mu_i')$ that subsumes or equals to $(T_i, \mu_i)$. For $i = 0$, the pair $(T_0, \mu_0)$ is in $\mathcal{M}^0$, according to Step 2 of the algorithm. Next, we assume that the hypothesis holds for $\mathcal{M}^i$ and show that it holds for $\mathcal{M}^{i+1}$.

We examine two cases. The first case is when $v_{i+1}$ is not a node of $T$. In this case, $(T_{i+1}, \mu_{i+1})$ is equal to $(T_i, \mu_i)$ because even if an edge is added to $T_i$ it is not an edge of $T$. If the pair $(T_i', \mu_i')$ is in $\mathcal{M}^{i+1}$ then $\mathcal{M}^{i+1}$ contains an element that subsumes $(T_{i+1}, \mu_{i+1})$. Otherwise, $\mathcal{M}^{i+1}$ contains a pair $(\tilde{T}, \tilde{\mu})$ that is created by extending $T_i'$ and $\mu_i'$. If so, $(\tilde{T}, \tilde{\mu})$ subsumes $(T_i', \mu_i')$ and, thus, subsumes $(T_{i+1}, \mu_{i+1})$.

In the second case, $v_{i+1}$ is a node of $T$. Let $T_{i+1}'$ be a tree that is produced by adding to $T_i'$ the node $v_{i+1}$ and the edge of $T$ that enters $v_{i+1}$. Since $v_0, v_1, \ldots, v_n$ are sorted topologically, $T'$ is a rooted tree. Let $\mu_{i+1}'$ be $\mu_i' \cup \{(v_{i+1}, \mu(v_{i+1}))\}$, i.e., an extension of $\mu_i'$ to $v_{i+1}$.

Because $\mu$ is a semiflexible matching of $T$, all the paths of $T$ satisfy the SF-condition w.r.t. $\mu$. Specifically, the path to $v_{i+1}$, in $T_{i+1}'$, is also a path in $T$. Hence, this path satisfies the SF-Condition w.r.t. $\mu_{i+1}$. So, $(T_{i+1}', \mu_{i+1}')$ is in an element of $\text{Ext}_{sf-or}(T_i', \mu_i', v_{i+1})$ and, consequently, it is an element of $\mathcal{M}^{i+1}$ or subsumed
by an element of $\mathcal{M}^{i+1}$. Finally, the induction hypothesis holds for $\mathcal{M}^{i+1}$, because $(T'_{i+1}, \mu'_{i+1})$ subsumes or is equal to $(T_{i+1}, \mu_{i+1})$. \hfill \Box

**Proposition 5.14 (Soundness)** Let $Q$ be a dag query and $D$ be a database. When computed over $Q$ and $D$, Algorithm SFORD produces only maximal semiflexible-or-matchings.

**Proof.** Suppose that $\mu$ is a matching in the result of Algorithm SFORD. In this case, according to Step 4 of the algorithm, there exists a tree $T$ such that $\mathcal{M}^n$ contains the pair $(T, \mu)$.

For each $i$ and every pair $(T_i, \mu_i)$ in $\mathcal{M}^i$, it holds that $\mu_i$ is a semiflexible matching of $T_i$ w.r.t. $D$. This is shown by induction on $i$. For $i = 0$, $\mathcal{M}^0$ contains only a single element comprising a mapping of the query root to the database root. Thus, the induction hypothesis holds. For $i > 0$, according to Definition 5.11, if $\mu_{i-1}$ is a semiflexible matching of $T_{i-1}$, then $\text{Ext}_{sfo}(T_{i-1}, \mu_{i-1}, v)$ consists of pairs $(T_i, \mu_i)$, such that $\mu_i$ is a semiflexible matching of $T_i$ w.r.t. $D$. This shows that $\mu$ is a semiflexible matching of $T$ w.r.t. $D$.

To see why $\mu$ is a maximal matching, consider the case where there exists $\mu'$—a semiflexible-or-matching of $Q$ w.r.t. $D$ that subsumes $\mu$. In this case, according to Proposition 5.13, $\mathcal{M}$ will include either $\mu'$ or a matching that subsumes $\mu'$. Since subsumed matching are removed, it follows that $\mu$ is not returned by the algorithm. This is a contradiction to our assumption. \hfill \Box

We now analyze the complexity of Algorithm SFORD. For each variable $v$ in $Q$, we denote by $\text{fan-in}(v)$ the number of edges that enter $v$ in $Q$.

**Proposition 5.15 (Complexity)** Let $Q$ be a dag query and $D$ be a database. The runtime of Algorithm SFORD is $O(d^2 + q^2mr(d + mr))$, where $q$ is the size of the query, $d$ is the size of the database, $m$ is the size of the result and $r$ is the value of the multiplication $\prod_{v \in Q, v \neq r} \text{fan-in}(v)$.

**Proof.** Computing the topological order over the nodes of $Q$, in Step 1, can be done in $O(q)$. Step 2 requires a constant time. In Step 3, there are $n$ iterations, where $n$
is the number of variables in \(Q\). In each iteration, the elements of \(M^i\) are extended. We will show later that there can be at most \(mr\) elements in \(M^i\).

An extension of a pair \((T, \mu)\), by adding a mapping to a variable \(v\), is computed in \(O(dq)\). In the extension, in the worst case, all the objects of \(D\) are tested. Testing if an object \(o\) can be assigned to \(v\) is by verifying that all the ancestors of \(v\) are mapped to objects that are connected to \(o\) by a path. In \(Q\) there are, at most, \(n\) ancestors of \(v\). We assume that it is in \(O(1)\) to test if a pair of objects is connected. To make this assumption true, we create, in a preprocessing step, a suitable data structure that tells which two database objects are connected by a path. Creating such a data structure can be done in \(O(d^2)\).

Removing subsumed elements from \(M^i\) requires \(O(q(mr)^2)\), since in \(M^i\) there are at most \(mr\) elements and because each subtree and each matching have an \(O(q)\) size. In total, the time complexity is \(O(q + d^2 + q(dm + q(mr)^2))\).

To complete the proof, we provide a bound on the number of elements in the sets \(M^i\). Consider a matching \(\mu\) in the result set \(M\). Let \(\mu_i\) be the restriction of \(\mu\) to \(v_0, \ldots, v_i\). We say that there is a repetition of size \(k\) \((k \geq 1)\) w.r.t. \(\mu\), in \(M^i\), if there are \(k + 1\) pairs \((T_1, \mu_i), \ldots, (T_{k+1}, \mu_i)\) in \(M^i\). That is, the restriction of \(\mu\) to the first \(i\) variables, in the topological sort, appears in \(k + 1\) different elements of \(M^i\).

Suppose that \(\mu\) is a matching in the result set \(M\). In \(M^0\) there are no repetitions w.r.t. \(\mu\), because \(M^0\) contains a single element. In \(M^1\) there is a repetition of size fan-in\((v_1)\) w.r.t. \(\mu\), because fan-in\((v_1)\) edges enter \(v_1\) in \(Q\). We continue by induction and see that, in \(M^i\), there is a repetition of size fan-in\((v_1) \cdot \ldots \cdot \text{fan-in}(v_i)\) w.r.t. \(\mu\). This shows that for each \(1 \leq i \leq n\), in \(M^i\), there is a repetition of size less than or equal to \(r\) w.r.t. each matching of the result. Thus, there are, at most, \(mr\) elements in \(M^i\).

\[\square\]

**Corollary 5.16 (Complexity for Tree Queries)** Let \(Q\) be a tree query and \(D\) be a database. The runtime of Algorithm SFORD is \(O(d^2 + q^2m(d + m))\), where \(q\) is the size of the query, \(d\) is the size of the database and \(m\) is the size of the result.
Proof. Since in a tree each node, other than the root, has exactly one incoming edge, the value of the expression $\prod_{v \in Q, v \neq r} fan-in(v)$ is 1. \qed

**Evaluating Cyclic Queries**

Evaluation of cyclic queries is similar to the evaluation of dag queries. The main principle is to iteratively construct all the subtrees of the query and compute the semiflexible matchings w.r.t. these trees.

Next, we describe Algorithm SFORG that computes semiflexible-or-matchings of queries that are arbitrary graphs. Consider a query $Q$ with $n+1$ variables and a database $D$. As in Algorithm SFORD, which computes semiflexible-or-matchings of dag queries, we create in SFORG sets of pairs $\mathcal{M}^0, \ldots, \mathcal{M}^n$. The set $\mathcal{M}^0$ consists of a pair $(T_0, \mu_0)$ where $T_0$ has just one node, the query root, and $\mu_0$ is the mapping of the query root to the database root. For each $i$, $\mathcal{M}^i$ is the set of all pairs $(T_i, \mu_i)$ such that $T_i$ is a subtree of $Q$ with exactly $i$ edges and $\mu_i$ is a semiflexible matching of $T_i$ w.r.t. the database $D$. Notice that there are $n$ sets since, for a query with $n+1$ nodes, a subtree of the query cannot have more than $n$ edges.

The sets $\mathcal{M}^1, \ldots, \mathcal{M}^n$ are created in $n$ iterative steps. In the $i$th iteration, the set $\mathcal{M}^i$ is created from the elements of $\mathcal{M}^{i-1}$ as follows. We create an empty set $\mathcal{M}^i$. Then, for each pair $(T_{i-1}, \mu_{i-1})$ in $\mathcal{M}^{i-1}$ and for each variable $v$, such that $v$ is not in $T_{i-1}$, the set $Ext_{sf-or}(T_{i-1}, \mu_{i-1}, v)$ is added to $\mathcal{M}^i$. Duplications are discarded.

During the creation of the sets $\mathcal{M}^1, \ldots, \mathcal{M}^n$, we create an additional set denoted $\mathcal{M}$. The set $\mathcal{M}$ consists of matchings that cannot be extended. At each step of creating a set $\mathcal{M}^i$, if a pair $(T_{i-1}, \mu_{i-1})$ is such that $Ext_{sf-or}(T_{i-1}, \mu_{i-1}, v)$ is empty for every $v$, then $\mu_{i-1}$ is added to $\mathcal{M}$. After the $n$th iteration, for each pair $(T_n, \mu_n)$ of $\mathcal{M}^n$, $\mu_n$ is added to $\mathcal{M}$. The final step of the algorithm is the creation of the matchings from the elements of $\mathcal{M}$. In this step, each matching $\mu$ in $\mathcal{M}$ is extended by mapping the undefined variables to null. Subsumed tuples and duplications are removed from $\mathcal{M}$ and $\mathcal{M}$ is returned.

Soundness and completeness of Algorithm SFORG are proven in a similar way to the proof of Algorithm SFORD.
Chapter 5  Combining Flexibility with Maximal Answers

**Proposition 5.17 (Time Complexity)** When computed over a query $Q$ and a database $D$, Algorithm SFORG has a time complexity $O(d^2 + q^3 d m r + q (m r)^2)$, where $q$ is the size of the query, $d$ is the size of the database, $m$ is the size of the result and $r$ is the value of the multiplication $\prod_{v \in Q, v \neq r_Q} (\text{fan-in}(v) + 1)$.

**Proof.** In $O(d^2)$ runtime, it is possible to construct a data structure that, for each pair of database objects, can answer in $O(1)$ time whether this pair of objects is connected by a path.

In the algorithm there are $q$ iterations. In each iteration, the elements of a set $\mathcal{M}^i$ are extended. There are at most $m r$ elements in $\mathcal{M}^i$ as will be proved later. For each pair $(T_i, \mu_i)$ in $\mathcal{M}^i$, there are at most $q$ edges that can extend $T_i$. For each edge $e$ that can be used to extend $T_i$, there are at most $d$ objects that can be assigned to the target node $v$ of $e$. Testing if an assignment of $o$ to $v$ satisfies the conditions of the semiflexible semantics requires to check that $o$ is on a path with all the objects that are assigned to ancestors of $v$. There can be at most $q$ ancestors to $v$. Thus, the runtime for a single iteration is $O(m r q d q)$ and $q$ iterations are computed in $O(q^3 d m r)$ time. Removing subsumed matchings, at the final step, is in $O((m r)^2 q)$ time.

We show why there are at most $m r$ elements in each set $\mathcal{M}^i$. First, note that there are no subsumed pairs in the set $\mathcal{M}^i$. This is because, for all the pairs in $\mathcal{M}^i$, all the trees in these pairs have exactly $i$ edges. Secondly, note that $\mathcal{M}^i$ is a set and does not include duplicate elements. Thirdly, for each pair $(T_i, \mu_i)$ in $\mathcal{M}^i$, the matching $\mu_i$ is a semiflexible matching of $T_i$. Thus, $\mu_i$ is either equal to a matching in the result or subsumed by a matching in the result. The last three observations show that it is sufficient to count the number of of pairs $(T_i, \mu_i)$ in $\mathcal{M}^i$, such that there is a semiflexible-or-matching of $Q$ w.r.t. $D$ that subsumes $\mu_i$.

Let $\mu$ be a semiflexible-or-matching of $Q$ w.r.t. $D$. We claim that the number of pairs $(T_i, \mu_i)$ in $\mathcal{M}^i$, such that $\mu$ subsumes $\mu_i$, is less than or equal to $r$. To prove this claim, let $r_Q, v_{j_1}, \ldots, v_{j_k}$ be the variables of $Q$ that $\mu$ maps to non-null values. Consider a subset $V_T$ of $\{r_Q, v_{j_1}, \ldots, v_{j_k}\}$ and let $T$ be a tree whose variables are $V_T$. For each $1 \leq l \leq k$, in $T$ there is at most one edge that enters $v_{j_l}$. There are
fan-in($v_{ji}$) edges that enter $v_{ji}$ in $Q$. So, in $T$, for each $1 \leq l \leq k$, either there is no edge that enters $v_{ji}$ or the edge that enters $v_{ji}$ is chosen from the fan-in($v_{ji}$) edges that enter $v_{ji}$ in $Q$. Hence, there are at most $\prod_{1 \leq l \leq k} (\text{fan-in}(v_{ji}) + 1)$ possibilities to choose the edges for a tree over $V_T$. For the case where $\mu$ maps all the variables of $Q$ to non-null values, the number of subtrees of $Q$ is $\prod_{v \in Q, v \neq r_Q} (\text{fan-in}(v) + 1)$, i.e., equals to $r$.

5.2.2 Computing Flexible-or Matchings and Flexible-Weak Matchings

In this section, we show how to compute flexible-or matchings and flexible-weak matchings in polynomial runtime, in the size of the input and output. The main principle behind the methods that we provide is to pose queries, under the or-semantics (weak semantics), to the reachability graph instead of posing the queries to the database itself.

In Theorem 3.55 we showed that the set of flexible matchings w.r.t. a database is equal to the set of rigid matchings with respect to the reachability graph. We now apply this principle to the combined semantics.

**Proposition 5.18 (Computation Reduction)** Let $Q$ be a query, $D$ be a database and $RG(D)$ be the reachability graph of $D$. The following two sets are equal.

- The set of flexible-or-matchings of $Q$ w.r.t. $D$.
- The set of or-matchings of $Q$ w.r.t. $RG(D)$.

**Proof.** Let $\mu$ be a matching of $Q$ w.r.t. $D$. Consider the following statements.

1. The matching $\mu$ is a flexible-or-matching of $Q$ w.r.t. $D$.
2. There exists a rooted fragment $F$ of $Q$, such that $\mu$ is a flexible matching of $F$ w.r.t. $D$.
3. There exists a rooted fragment $F$ of $Q$, such that $\mu$ is a rigid matching of $F$ w.r.t. $RG(D)$. 


Chapter 5  Combining Flexibility with Maximal Answers

4. The matching \( \mu \) is an or-matching of \( Q \) w.r.t. \( RG(D) \).

According to Definition 5.3, Statement 1 is true if and only if Statement 2 is true. Statement 2 is true if and only if Statement 3 is true, due to Theorem 3.55. Statement 3 and Statement 4 are either both true or both false, according to Proposition 5.2. \( \square \)

For weak queries, the next proposition describes a principle that is similar to the the principle of Proposition 5.18.

**Proposition 5.19 (Computation Reduction)** Let \( Q \) be a query, \( D \) be a database and \( RG(D) \) be the reachability graph of \( D \). The following two sets are equal.

- The set of flexible-weak matchings of \( Q \) w.r.t. \( D \).

- The set of weak-matchings of \( Q \) w.r.t. \( RG(D) \).

The proof of Proposition 5.19 is similar to the proof of Proposition 5.18.

Let \( Q \) be a query and \( D \) be a database. Computing flexible-or-matchings (flexible-weak matchings) of \( Q \) w.r.t. \( D \) is a two-step process. Firstly, \( RG(D) \)—the reachability graph of \( D \)—is constructed. Secondly, \( Q \) is evaluated under the or-semantics (weak-semantics) w.r.t. \( RG(D) \). Correctness follows from Proposition 5.18 (Proposition 5.19).

**Theorem 5.20 (Polynomial Time Complexity)** Let \( Q \) be a query and \( D \) be a database. There exists an algorithm that computes the flexible-or-matchings of \( Q \) w.r.t. \( D \) in polynomial time in the size of the input and output. This theorem is also true if we replace “flexible-or-matchings” with “flexible-weak-matchings”.

**Proof.** The theorem follows from the following two facts. First, the size of the reachability graph is polynomial in the size of the database. Secondly, as shown in Chapter 3, there exists an algorithm that computes or-matchings (weak-matchings) in polynomial time in the size of the input and output. \( \square \)
Table 5.3: The complexity of query evaluation under the four combined semantics. (NP-Complete results refer to checking non-emptiness.)

## 5.3 Summary of Contributions

In this chapter, we presented four new semantics, namely, the semiflexible-or, the semiflexible-weak, the flexible-or and the flexible-weak semantics. Query evaluation was investigated under these four new semantics. It was shown that under the semiflexible-or semantics, a tree query can be evaluated in polynomial time in the size of the input and the output. For a DAG query, there is no polynomial-time algorithm, in the size of the input and the output, unless P = NP. Under the flexible-or and the flexible-weak semantics, any arbitrary query can be evaluated in polynomial time in the size of the input and the output. When the query is a tree, the semiflexible-weak semantics and the semiflexible-or semantics are the same. Thus, under the semiflexible-weak semantics, tree queries can be computed in polynomial time in the size of the input and the output. For DAG queries and cyclic queries it is still an open question whether such an algorithm exists.
Chapter 6

Full Disjunctions

In this chapter, we present the full-disjunction approach for oblivious querying. The manner of oblivious querying, in the previous chapters, required from users to provide a query in the form of a rooted graph. Since providing a rooted graph could be cumbersome, in the full disjunction approach, the semistructured database is automatically transformed into a relation and the user queries this relation using a traditional relational query language. Thus, the user is oblivious to the structure of the semistructured database.

The outline of this chapter is as follows. In Section 6.1 a short introduction is given. Section 6.2 provides some preliminary definitions and, in particular, full disjunctions are defined. Section 6.3, explains how to transform a semistructured database into a universal relation. Such a transformation facilitates the querying of semistructured databases and admits the querying of a single relation with a known schema—instead of querying a graph whose schema is unknown or is more complicated. In Section 6.4, it is shown how to reduce the evaluation of full disjunctions to the evaluation of maximal weak matchings. This reduction provides an algorithm, for full disjunctions, that has polynomial-time input-output complexity. Section 6.5 discusses projections of a full disjunction on a given set of attributes. It is shown that for the case of relations with γ-acyclic schemas (see Appendix A, for a definition of γ-acyclic hypergraphs), there is an algorithm that computes the projection of a full disjunction in polynomial time in the size of the input and the output. However, in the general case, i.e., when the relation schemas are not γ-acyclic, such algorithm
does not exists, unless $P = NP$. Section 6.6 presents a generalization of full disjunctions that facilitates integration of relational data by means of join conditions that are more general than just equalities. Finally, in Section 6.7, we discuss the contribution of this chapter and conclude.

6.1 Motivation

For a start, we explain the motivation behind full disjunctions. The usual way to integrate data from several relations is by computing a natural join of the relations. However, in a natural join, there could be dangling tuples. Given a set of relations $r_1, \ldots, r_n$, a tuple $t$, in some relation $r_i$ with schema $R_i$, is a dangling tuple if the the projection of the natural join of $r_1, \ldots, r_n$ on the attributes of $R_i$ does not include $t$. Obviously, in a natural join, dangling tuples are not part of the result. That is, they do not appear as part of any tuple of the result. In this sense, natural joins cause a information loss.

Consider a set of relations $\mathcal{R} = \{r_1, \ldots, r_n\}$. Suppose that we take all the subsets of $\mathcal{R}$, compute the natural join w.r.t. each subset and add the tuples of these natural joins to a set $\mathcal{J}$. In $\mathcal{J}$, no dangling tuple is lost. However, there are two fundamental problems in $\mathcal{J}$ that prevent $\mathcal{J}$ from being useful.

One problem with $\mathcal{J}$ is that tuples in the result may come from disconnected relations. We consider two relations as directly connected if their schemas have a non-empty intersection. Two relations $r$ and $r'$ are connected if one of the following two conditions is true: (1) $r$ and $r'$ are directly connected; (2) there are two connected relations $\bar{r}$ and $\bar{r}'$ such that $\bar{r}$ is directly connected to $r$ and $\bar{r}'$ is directly connected to $r'$. Joining disconnected tuples should be avoided. Otherwise, we would not be able to distinguish between the next two cases. First, a case where two pieces of information are in one tuple of the result because they are related. Second, a case where the two pieces of information appear in one tuple of the result because they come from disconnected tuples.

A second problem with $\mathcal{J}$ is that it contains tuples that are not maximal. More specifically, consider a tuple $t$ that is the join of the tuples $t_{i_1}, \ldots, t_{i_k}$, where
Chapter 6  Full Disjunctions

t_{i_1}, \ldots, t_{i_k} are tuples of the relations r_{i_1}, \ldots, r_{i_k}, respectively. If t is in \mathcal{J}, then for each subset of t_{i_1}, \ldots, t_{i_k}, the join of the tuples in this subset will also be in \mathcal{J}. This may cause an exponential blowup in the size of \mathcal{J}, without adding information.

In an integration of two relations, the outerjoin can give an answer to the three problem that were presented above: loss of dangling tuples, connecting unrelated pieces of data and an exponential blowup due to keeping non-maximal elements. An outerjoin of two relations r_1 and r_2 is a relation that consists of all the tuples in the natural join of r_1 and r_2. In addition, the outerjoin contains all the tuples of r_1 (r_2) that are not joined with any tuple of r_2 (r_1), padded with null values.

For integration of more than two relations, the semantics of outerjoins is problematic, since outerjoins are not associative. Galiando-Legaria [34] proposed full disjunctions, which are commutative and associative, as an alternative to outerjoins. Essentially, the full disjunction of a set of relations is obtained by taking all joins of any subsets of relations, excluding joins that involve a Cartesian product, and removing subsumed tuples.

In full disjunctions, dangling tuples are not discarded. In addition, the two problems that were discussed above are prevented—the problem of tuples that are produced by a join of disconnected tuples and the problem of causing an exponential blowup due to keeping non-maximal tuples in the result.

Rajaraman and Ullman [55] showed that the full disjunction of some given relations can be evaluated by a natural outerjoin sequence if and only if the relation schemas are connected and \gamma-acyclic. It is easy to see that the sequence of outerjoins has a polynomial input-output complexity. They did not provide a polynomial time algorithm, under input-output complexity, for computing full disjunction of relations with schemas that are not \gamma-acyclic.

In the next section, we provide a formal definition of full disjunctions. We describe how to use full disjunction for oblivious querying of semistructured data and we present an algorithm that computes full disjunctions in polynomial time in the size of the input and output. The runtime of the algorithm remains polynomial even for relations with schemas that are not \gamma-acyclic.
Chapter 6  Full Disjunctions

6.2 Preliminaries

Let \( r_1, \ldots, r_n \) be relations with relation schemas \( R_1, \ldots, R_n \), respectively. The \( j \)th tuple of \( r_i \) is denoted as \( t_{ij} \). Given a subset \( X \subseteq R_i \), the projection of the tuple \( t_{ij} \) on \( X \) is denoted as \( t_{ij}[X] \).

Two distinct relation schemas \( R_i \) and \( R_j \) are connected if \( R_i \cap R_j \) is non-empty. This definition is generalized to \( m \) (\( m \geq 1 \)) distinct relation schemas as follows. The schema graph of \( R_{i_1}, \ldots, R_{i_m} \) consists of a node for each \( R_{i_j} \) and edges in both directions between \( R_{i_h} \) and \( R_{i_k} \) (\( 1 \leq h < k \leq m \)) if \( R_{i_h} \) and \( R_{i_k} \) are connected, i.e., \( R_{i_h} \cap R_{i_k} \) is non-empty. We say that the relation schemas \( R_{i_1}, \ldots, R_{i_m} \) are connected if their schema graph is connected. (There is an equivalent definition of connectivity that is based on the notion of the hypergraph of \( R_{i_1}, \ldots, R_{i_m} \).) We say that tuples \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \) from \( m \) (\( m \geq 1 \)) distinct relations are connected if their relation schemas are connected. Note that a single tuple \( t_{i_1 j_1} \) is connected.

Two connected tuples \( t_{i_1 j_1} \) and \( t_{i_2 j_2} \) (\( i_1 \neq i_2 \)) are join consistent if \( t_{i_1 j_1}[R_{i_1} \cap R_{i_2}] = t_{i_2 j_2}[R_{i_1} \cap R_{i_2}] \). More generally, \( m \) tuples \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \) from \( m \) distinct relations are join consistent if every pair of connected tuples \( t_{i_h j_h} \) and \( t_{i_k j_k} \) is join consistent. The natural join of \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \), denoted \( \mathcal{N}_{k=1}^m t_{i_k j_k} \), is either empty or has one tuple over the attributes of \( \cup_{k=1}^m R_{i_k} \). It has one tuple if and only if \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \) are join consistent.

A universal tuple is defined over the set of all the attributes, namely \( \cup_{i=1}^n R_i \), and its columns are filled with nulls as well as non-null values. If \( u \) is non-null exactly on the attributes of \( Z \), then \( u[Z] \) is called the non-null portion of \( u \) and is denoted as \( \hat{u} \). The universal tuple \( u \) is called an integrated tuple if there are \( m \geq 1 \) connected and join-consistent tuples \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \), such that \( \hat{u} = \mathcal{N}_{k=1}^m t_{i_k j_k} \). The tuples \( t_{i_1 j_1}, \ldots, t_{i_m j_m} \) are called generators of \( u \).

A universal tuple \( v \) subsumes a universal tuple \( u \), denoted \( u \subseteq v \), if \( v[Z] = u[Z] \), where \( u \) is non-null exactly on \( Z \). Given a set \( \mathcal{D} \) of universal tuples, \( u \in \mathcal{D} \) is maximal if it is not subsumed by any other tuple of \( \mathcal{D} \).

**Definition 6.1 (Full Disjunction)** The full disjunction of the relations \( r_1, \ldots, r_n \) is the set of all maximal integrated tuples that can be generated from \( m \) (\( 1 \leq m \leq n \))
Figure 6.1: The full disjunction of four relations.

tuples of \( r_1, \ldots, r_n \).

**Example 6.2** In Figure 6.1, the full disjunction of four relations is depicted. The first tuple of the full disjunction is created as a join of the the first tuples of the relations \( R_1 \) and \( R_2 \). The second tuple of the full disjunction is the join of the second tuple of \( R_1 \), the first tuple of \( R_3 \) and the second tuple of \( R_4 \). The third tuple is a join of the first tuples of \( R_3 \) and \( R_4 \). Finally, the fourth tuple is created from the second tuple of \( R_3 \).

### 6.3 Transforming Semistructured Data to Relational

We now show how full disjunctions can be used for oblivious querying of semistructured databases, provided that in the database there are no repeating labels on simple paths. To facilitate querying of irregular semistructured data, the database is transformed into a relation. Intuitively, the transformation is as follows. First, for each simple database path, from the root to a leaf, a tuple is created. Secondly, the tuples are divided to relations according to the labels on the paths that define them. Finally, the full disjunction of the relations is computed and returned. Next, we provide some definitions and then introduce the transformation.

**Definition 6.3 (Induced Schema and Induced Tuple)** Let \( D \) be a semistructured database and \( \phi = o_0l_1o_1 \cdots o_{n-1}l_no_n \) be a simple path from the root of \( D \) to a leaf \( o_n \). Furthermore, assume that there are no repeated labels on \( \phi \). The induced
schema of \( \phi \) is the set \( \text{Sch}_\phi \) that consists of the labels on \( \phi \), i.e., \( l_1, \ldots, l_n \). The induced tuple of \( \phi \) is a tuple \( t_\phi \) over the schema \( \text{Sch}_\phi \). For \( 1 \leq i \leq n - 1 \), \( t_\phi[l_i] \) is the oid of \( o_i \). For the atomic node \( o_n \), \( t_\phi[l_n] \) is the atomic value that is attached to \( o_n \), in \( D \).

Consider a semiflexible database \( D \) that has no repeated labels on simple paths. The method for transforming \( D \) into a relation is as follows.

**Transform a Semistructured Database to a Relation**

1. Let \( \mathcal{R} \) be an empty set of relations.
2. Compute all the simple paths, from the root to a leaf, in \( D \).
3. For each path \( \phi \) that was found in Step 1, compute the induced schema of \( \phi \), \( \text{Sch}_\phi \), and the induced tuple of \( \phi \), \( t_\phi \). If there exists a relation \( r \) in \( \mathcal{R} \) with the schema \( \text{Sch}_\phi \), then add \( t_\phi \) to \( r \). Else, create a new relation \( r \) with the schema \( \text{Sch}_\phi \), add \( t_\phi \) to \( r \) and add \( r \) to \( \mathcal{R} \).
4. Compute \( \mathcal{F} \), the full disjunction of the relations of \( \mathcal{R} \) and return \( \mathcal{F} \).

**Example 6.4** In this example we demonstrate the full-disjunction approach by applying the transformation to the movie database of Figure 2.1. The tables that are created in the first step of the transformation are presented in Appendix B. The result of the transformation is shown in Table 6.1.

Without providing any information, the user receives a table that she can query using a relational query language. Note that a projection on the attributes “Movie” and “Actor” produces all the movie-actor pairs in which the actor acted in the movie.

The full disjunction approach is less accurate than the combined semantics that were presented in Chapter 5. By this we mean that there can be tuples in \( \mathcal{F} \) where the tuples indicate that two values or objects are related, while actually these values or object are not related. This happens, for example, in Line 6 of the table. In this line, the T.V. Series “Twin Peaks” is attached to the year 1984, although 1984 is actually the year when “Dune” was produced. Thus, the full disjunction approach
Table 6.1: The result of transforming the movie database into a relation, using full disjunction.

is suitable for naive users but it may have errors that the flexible-weak semantics (or the other combined semantics) of Chapter 5 would not have.

6.4 Computing Full Disjunctions

In order to compute the full disjunction, we need to view tuples as objects and relation schemas as variables. By a slight abuse of notation, $t_{ij}$ denotes both the tuple and its oid. Similarly, $R_i$ denotes both the relation schema and its corresponding variable.

We construct a database $D = (O, E_D, r_D, \alpha)$ and $n$ queries $Q_i = (V, E_{Q_i}, r_{Q_i})$ ($1 \leq i \leq n$) as follows. $D$ has a root $r_D$ and an object for each tuple $t_{ij}$. There are edges in both directions between every pair of connected and join-consistent tuples. There is also an edge from the root to every tuple. An edge that enters a tuple of $r_k$ ($1 \leq k \leq n$) is labeled with $l_k$. The values of the atomic objects can be chosen arbitrarily.

The query $Q_i = (V, E_{Q_i}, r_{Q_i})$ is constructed from the schema graph of $R_1, \ldots, R_n$ by adding a root $r_{Q_i}$ and an edge from the root to each $R_i$. An edge that enters $R_k$ ($1 \leq k \leq n$) is labeled with $l_k$. Note that only one connected component of the schema graph of $R_1, \ldots, R_n$ can be reached from the root of $Q_i$. 
Consider a weak matching \( \mu \) of some \( Q_i \) w.r.t. \( D \). Recall that \( \mu \) assigns to each variable of \( Q_i \) either a tuple or null. Let \( g(\mu) \) denote the tuples in the image of \( \mu \). Since \( \mu \) is a weak matching, the tuples of \( g(\mu) \) are connected and join consistent. Therefore, the tuples of \( g(\mu) \) generate an integrated tuple that is denoted as \( \triangleright \sigma(\mu) \).

**Proposition 6.5 (Soundness and Completeness)** The tuple \( u \) is an integrated tuple of \( r_1, \ldots, r_n \) if and only if there is a weak matching \( \mu \) of some \( Q_i \) w.r.t. \( D \), such that \( u = \triangleright \sigma(\mu) \).

**Proof.** As noted earlier, if \( \mu \) is a weak matching of \( Q_i \) w.r.t. \( D \), then \( \triangleright \sigma(\mu) \) is an integrated tuple. For the other direction, consider an integrated tuple \( u \) that is generated by the tuples \( t_{i_1,i_1}, \ldots, t_{i_m,j_m} \). Let \( \mu \) be the assignment for \( Q_i \) w.r.t. \( D \) that is defined as follows: \( \mu(r_{Q_i}) = r_D \), \( \mu(R_{i_k}) = t_{i_k,j_k} \) (1 \( \leq k \leq m \)), and \( \mu(R_i) = \perp \) for any \( R_i \) that is not among \( R_{i_1}, \ldots, R_{i_m} \). Note that all the edges that enter \( R_{i_k} \) and all the edges that enter \( t_{i_k,j_k} \) (1 \( \leq k \leq m \)) have the same label \( l_{i_k} \). Moreover, \( t_{i_1,i_1}, \ldots, t_{i_m,j_m} \) are connected and join consistent. Thus, \( \mu \) is a weak matching. By the construction of \( \mu \), we have that \( u = \triangleright \sigma(\mu) \). This proves the proposition. \( \square \)

The full disjunction \( \mathcal{F} \) of \( r_1, \ldots, r_n \) can be computed as follows. First, compute the set \( \mathcal{M}_w(Q_i, D) \) for \( i = 1, 2, \ldots, n \). Let \( \mathcal{M} \) be obtained from \( \cup_{i=1}^n \mathcal{M}_w(Q_i, D) \) by removing subsumed matchings.\(^1\) Finally, \( \mathcal{F} = \{ \triangleright \sigma(\mu) \mid \mu \in \mathcal{M} \} \).

**Theorem 6.6 (Complexity)** The set \( \mathcal{F} \) is the full disjunction of the relations \( r_1, \ldots, r_n \) and it can be computed in \( O(n^5 s^2 f^2) \) time, where \( n \) is the number of relations, \( s \) is the total size of all the relations and \( f \) is the size of \( \mathcal{F} \).

**Proof.** From Proposition 6.5, it follows that \( \mathcal{F} \) consists of integrated tuples. To complete the proof that \( \mathcal{F} \) is the full disjunction, it should be observed that if \( \mu_1 \) and \( \mu_2 \) are two assignments of \( Q_i \) w.r.t. \( D \), such that \( \mu_1 \sqsubseteq \mu_2 \), then \( \triangleright \sigma(\mu_1) \sqsubseteq \triangleright \sigma(\mu_2) \). Therefore, it is sufficient to compute only the maximal weak matchings in order to generate all the maximal integrated tuples.

\(^1\) \( \mathcal{M}_w(Q_i, D) \) has matchings of all the \( Q_i \), but all these matchings are for the same set of variables (ignoring the root) and therefore, subsumptions among them are well defined.
**Chapter 6  Full Disjunctions**

In order to show that the running time of the algorithm is polynomial in the size of the input and the output, it is necessary to show that \( \mathcal{F} \) has only maximal integrated tuples. We will derive a contradiction by assuming that there are two distinct integrated tuples \( u_1 \) and \( u_2 \) in \( \mathcal{F} \), such that \( u_1 \subseteq u_2 \). Let \( \mu_1 \) and \( \mu_2 \) be two maximal weak matchings of \( Q_i \) and \( Q_j \), respectively, such that \( u_1 = \varphi(\mu_1) \) and \( u_2 = \varphi(\mu_2) \). Essentially, it can be shown that there is a weak matching \( \mu \) of \( Q_j \) w.r.t. \( D \), such that \( u_2 = \varphi(\mu) \) and \( \mu \) subsumes both \( \mu_1 \) and \( \mu_2 \). This contradicts the fact that \( \mu_1 \) and \( \mu_2 \) are maximal in \( \mathcal{M} \).

The size of each query \( Q_i \) is \( O(n^2) \) and the size of the database \( D \) that is constructed from the relations is \( s^2 \). The size of each \( \mathcal{M}_w(Q_i, D) \) is no more than the size of \( \mathcal{F} \). In general, the size of the product graph is \( O(pd) \), where \( p \) is the size of the query and \( d \) is the size of the database. However, in the computation of the full disjunction, for each node \( t_{ij} \) of \( D \), the product graph \( Q_k \times D \) \( (1 \leq k \leq n) \) has at most one node \( (R_i, t_{ij}) \in Q_k \times D \) that is reachable from the root. Consequently, the size of the portion of \( Q_k \times D \) that is reachable from the root is \( O(d) \). Therefore, the size of the stratum traversal is \( O(pd) \) and the time to compute each \( \mathcal{M}_w(Q_i, D) \) is \( O(n^4 s^2 f^2) \). Hence, \( \mathcal{M} \) is computed in \( O(n^5 s^2 f^2) \) time. Removing subsumed matchings from \( \bigcup_{i=1}^n \mathcal{M}_w(Q_i, D) \) takes \( O(n^3 f^2) \) time. The final step of constructing \( \mathcal{F} \) from \( \mathcal{M} \) takes \( O(f) \) time. Thus, the running time is \( O(n^5 s^2 f^2) \).

Theorem 6.6 shows that full disjunction can be computed in polynomial runtime, under input-output complexity. However, in the polynomial expression, the exponent of \( n \) is 5, where \( n \) is the number of relations in the input. Hence, in order to efficiently compute the full disjunction in the case where the input comprises many relations, optimization techniques should be applied. Next, we present an optimization rule that allows to improve the evaluation runtime for many cases of full disjunctions. We start by providing some definitions.

Consider a connected undirected graph \( G \). A separation of \( G \) w.r.t. a node \( v \) are two subgraphs of \( G \), \( G_1 \) and \( G_2 \), such that the following holds. (1) \( v \) is a node in both \( G_1 \) and \( G_2 \). (2) Every node of \( G \), except \( v \), is either in \( G_1 \) or in \( G_2 \) but not in both. (3) For each two nodes \( v_1 \) and \( v_2 \) of \( G_1 \) and \( G_2 \), respectively, there is no
edge between \( v_1 \) and \( v_2 \) in \( G \), unless \( v_1 \) is \( v \) or \( v_2 \) is \( v \). We say that \( G_1 \) and \( G_2 \) are a non-trivial separation of \( G \) if they are a separation in which each of the two graphs has at least two nodes.

An articulation node of \( G \) is a node whose removal will disconnect \( G \). That is, a node \( v \), in \( G \), is an articulation node if removing \( v \) from \( G \) (and also removing from \( G \) all the edges that are incident on \( v \)) creates a subgraph of \( G \) that is not connected. It is easy to see that there exists a non-trivial separation of \( G \) w.r.t. \( v \) if and only if \( v \) is an articulation node of \( G \). Computing all the articulation nodes of \( G \) is in \( O(|E|) \) where \( |E| \) is the number of edges in \( G \) (see [27] for details).

We provide some required notations. Consider a set of relations \( r_1, \ldots, r_n \) with schemas \( R_1, \ldots, R_n \), respectively. Let \( G \) be the schema graph of \( R_1, \ldots, R_n \). By \( FD(G) \) we denote the full disjunction of \( r_1, \ldots, r_n \), i.e., the full disjunction of the relations that correspond to the schemas that are nodes of \( G \). We use the symbol \( \bowtie \) to denote the outerjoin operation. The next proposition shows how graph separations can assist in computing full disjunctions.

**Proposition 6.7 (Applying an Outerjoin)** Suppose that \( r_1, \ldots, r_n \) are relations with schemas \( R_1, \ldots, R_n \), respectively. Let the schema graph of \( R_1, \ldots, R_n \) be \( G \) and \( R_j \) be an articulation node of \( G \). In addition, let \( G_1 \) and \( G_2 \) be a separation of \( G \) w.r.t. \( R_j \). Then \( FD(G_1) \bowtie FD(G_2) \) is equal to \( FD(G) \).

The proof of Proposition 6.7 is given in Appendix C.

Based on Proposition 6.7, we propose the Separate-and-Join method for computing full disjunctions. Consider a set of relations \( r_1, \ldots, r_n \) with schemas \( R_1, \ldots, R_n \), respectively. Separate-and-Join works recursively. First, the schema graph \( G \) of \( R_1, \ldots, R_n \) is computed. If there is a separation \( G_1 \) and \( G_2 \) of \( G \) then Separate-and-Join is called with \( G_1 \) and with \( G_2 \). The results of these calls are joined using an outerjoin. The result of the outerjoin is returned. If there is no separation of \( G \) then the general evaluation algorithm (the one that is presented at the beginning of the section) is called.

Note that at the worst case, Separate-and-Join has the same runtime complexity as the general algorithm. However, in many cases, evaluation of a full disjunction
by Separate-and-Join is performed as series of outerjoin operations.

6.5 Projections and Restrictions of Full Disjunctions

One way of generalizing the result of Theorem 6.6 is by considering the complexity of evaluating a projection of the full disjunction. Formally, the projection problem is the problem of computing the projection of the full disjunction on a given set of attributes $X$.

In some cases, it might be desirable to compute only those tuples of the full disjunction that are non-null on all the attributes of a given set $X$. Formally, the restriction problem is the problem of computing just those tuples of the full disjunction that are non-null on $X$. The following theorem shows that the restriction problem cannot be computed in polynomial time in the size of the input and the output, even if the relation schemas are $\alpha$-acyclic (a definition of $\alpha$-acyclic schemas is provided in Appendix A) and $X$ has only two attributes.

**Theorem 6.8 (NP-Completeness)** Let $r_1, \ldots, r_n$ be relations with $\alpha$-acyclic relation schemas and let $X$ be a set of two attributes. Deciding whether the full disjunction has a tuple that is non-null on all the attributes of $X$ is NP-complete.

**Proof.** NP-hardness is shown by a reduction of the Hamiltonian-path problem. Let $G = (V, E)$ be a given directed graph, and let $s$ and $t$ be two nodes in $G$. The problem is to decide whether $G$ has a directed path from $s$ to $t$ that goes through all the nodes of $G$, visiting each node exactly once.

Suppose that $G$ has $k$ nodes, denoted $n_1, \ldots, n_k$, and $m$ edges. We assume that $s = n_1$ and $t = n_k$. We construct the following $m + 2$ relations:

1. A relation $r_s$ that has the relation schema $S(A, N_1)$ and contains the single tuple $(1, 1)$.
2. A relation $r_t$ that has the relation schema $T(N_k, B)$ and contains the single tuple $(k, k)$. 
Chapter 6  Full Disjunctions

3. For each edge $e_l = (n_{i_l}, n_{j_l})$ (1 ≤ $l$ ≤ $m$), we construct a relation $r_l$ that has the relation schema $E_l(N_{i_l}, N_{j_l})$ and contains the $k - 1$ tuples: (1,2), (2,3), ..., (k - 1, k).

In addition to the above $m + 2$ relations, we also construct an empty relation $r_\alpha$ that has the set of all the attributes as its relation schema. The sole purpose of $r_\alpha$ is to ensure that the relation schemas of the $m + 3$ relations are $\alpha$-acyclic.

Let $\mathcal{F}$ denote the full disjunction of the $m+3$ relations and let $q$ be the projection of $\mathcal{F}$ on the attributes $A$ and $B$.

It is easy to see that only the tuples $(1,k)$, $(1,\bot)$ and $(\bot,k)$ may appear in $q$. Furthermore, there is a Hamiltonian path in $G$ from $s$ to $t$ if and only if the tuple $(1,k)$ is in the result of $q$. This shows NP-hardness.

To show membership in NP, note that we can guess an assignment of values and nulls to all the attributes of all the relation schemas, such that the attributes of $X$ are assigned non-null values. We can then verify in polynomial time that the guess creates an integrated tuple.

The reduction that is used in the proof of Theorem 6.8 implies that the projection problem cannot be computed in polynomial time under input-output complexity if $P \neq \text{NP}$. For the problem of deciding whether a tuple is in the projection of the full disjunction on a given set of attributes, membership in NP is also shown as described in the proof of Theorem 6.8. Thus, we have the following corollary.

**Corollary 6.9 (NP-Completeness)** Deciding whether a given tuple is in the projection of the full disjunction on $X$ is NP-complete.

Theorem 6.8 and Corollary 6.9 show that if $P \neq \text{NP}$, then there cannot exist an algorithm, for either the restriction problem or the projection problem, that is polynomial under input-output complexity. It is easy to see that the projection problem and the restriction problem have polynomial-time algorithms if each relation schema either contains $X$ or is disjoint from $X$. Note that, in particular, this condition is satisfied if $X$ is a singleton.

A related problem to the projection problem is the evaluation of a projection of the natural join of $n$ relations. Yannakakis [65] showed that this problem has a
polynomial-time algorithm, in the size of the input and the output, if the relation schemas are \( \alpha \)-acyclic. In the general case, deciding non-emptiness of the natural join of \( n \) relations is NP-Complete [49].

In [55], Rajaraman and Ullman showed that the full disjunction of \( r_1, \ldots, r_n \) can be evaluated by a natural outerjoin sequence if and only if the relation schemas are connected and \( \gamma \)-acyclic. Their result can be generalized as follows.

**Theorem 6.10 (Complexity)** The projection problem has a polynomial-time algorithm under input-output complexity if the relation schemas are \( \gamma \)-acyclic.

The proof of Theorem 6.10 is given in Appendix C.

### 6.6 Generalizing Full Disjunctions

Full disjunction are based either on equijoins [34] or natural joins [55]. Quit frequently, however, the process of integrating information involves more general conditions than merely equality of attributes. Consider the scenario of piecing together information about people, in the absence of a unique ID for each person. Joining two records, simply because both carry the same name, might be error prone. A safer approach could be to join two records if the names are the same or similar (e.g., one has a middle initial and the other does not) and either the address or one of the phone numbers (e.g., home, office or mobile phone) are the same. A similar scenario is joining two records, such that one has a complete address while the other only contains the city and state. In this case the appropriate condition is that the city and state from the second record appear in the address of the first record.

The approach of Section 6.4 can be easily generalized to the above scenarios. We use the same queries as in Section 6.4 and use the same database, but the edges should reflect the new conditions. Thus, there are edges in both directions between \( R_i \) and \( R_j \) if some condition is specified for this pair of relation schemas. Similarly, there are edges in both directions between the tuples \( t_{i_1, j_1} \) and \( t_{i_2, j_2} \) if these two tuples satisfy the condition that is specified for their relation schemas, \( R_i \) and \( R_{i_2} \).

Conditions of general types could be used for joining tuples. Essentially, we may
assume that queries are formulated using a SELECT-FROM-WHERE clause and the
WHERE clause is a conjunction of conditions $C_1 \land \ldots \land C_k$, where each $C_h$ involves
either one or two relations. If $C_h$ involves just one relation $r_i$, then it is a selection
that can be applied to $r_i$ before starting the evaluation of the full disjunction. If $C_h$
involves two relations $r_i$ and $r_j$, then it may be any condition that can be evaluated
for pairs of tuples from these two relations.

Thus, the weak semantics provides the means to generalize full disjunctions and
thereby facilitating integration of information from the Web, as well as from other
heterogeneous sources, in a more general manner than earlier work.

6.7 Summary of Contributions

In Chapter 6, we have shown that evaluation of full disjunctions is reducible to
evaluation of queries under the weak semantics. Thus, full disjunctions can be
computed in polynomial time in the size of the input and the output. Previously,
the only known algorithm to compute full disjunctions in the general case was by
evaluating all connected joins and removing subsumed tuples. That algorithm runs
in exponential time in the size of the input and the output. Hence, our result makes
full disjunctions practical for real-world applications.

For $\alpha$-acyclic relation schemes, Yannakakis [65] showed that any projection of the
natural join of all the relations can be computed in polynomial time in the size of the
input and the output. Theorem 6.10 builds on the results of [55, 65]. We have also
shown that when the relation schemes are $\alpha$-acyclic, there are no polynomial-time al-

The weak semantics provides a substantial generalization of full disjunctions,
by allowing general types of join conditions (provided that those conditions can
be computed efficiently). This is achieved without increasing the time complexity
of evaluating full disjunctions, in stark contrast to other approaches for querying
incomplete information, where complex join conditions quickly lead to intractability
of query evaluation.
Chapter 6  Full Disjunctions

The OR-semantics can provide another, more general form of full disjunctions. In the OR-semantics, an answer should be connected, but it does not have to satisfy all the join conditions among its parts; instead, it is sufficient to satisfy only a subset of the conditions, provided that the satisfied join conditions are connected. In traditional databases, this would not be an answer at all, but in the realm of the Web this might still be of interest to the user, although it would probably be ranked lower than answers that are obtained under the weak semantics.
Chapter 7

Conclusion

Our work introduces novel ways of querying semistructured data and XML. We propose new querying methods that are ontology-based querying—the user is given the ontology of the database, i.e., a list of labels, and is not required to have any knowledge about structural details of the data. An ontology-based querying is well suited to querying data in the following cases: (1) the data do not conform to a schema, (2) the data have a structure that changes frequently, and (3) the schema of the data is unknown. For all these cases, traditional querying methods are generally problematic since the user formulates a query having a specific schema in mind and receives a result that does not include all the expected answers.

Irregularity and incompleteness are two fundamental features of semistructured data and ontology-based querying deals with both. In irregular databases, the order of labels, in database paths, is unknown. The paradigm of flexible queries introduced in this work facilitates querying of data that are irregular. In incomplete databases, some fragments of the database are expected to include certain labels, but some of these labels are missing. The paradigm of queries with maximal answers that was introduced in [41, 42] facilitates querying of data that are incomplete.

The main contributions of our work is by proposing methods for dealing with irregularity and incompleteness in semistructured data and XML. In the paradigm of flexible queries, we defined the semiflexible and the flexible semantics. Query evaluation, under these semantics, was investigated. We also investigated query equivalence and the novel concept of database equivalence. In the framework of queries with maximal answers, we continued the work of Kanza, Nutt and Sagiv
Chapter 7 Conclusion

[41, 42] by providing a polynomial-time algorithm, in the size of the input and the output, for computing cyclic queries, under the weak and the or-semantics. We also showed how to combine the semantics of the two paradigms. The combined semantics facilitate the querying of data that are both irregular and incomplete. The user queries the data while being oblivious to the structural details of the data and to the possibility of incompleteness.

An additional important contribution of our work is by showing that evaluation of full disjunctions is reducible to evaluation of queries under the weak semantics. Thus, full disjunctions can be computed in polynomial time in the size of the input and the output. Previously, known algorithms for computing full disjunctions of any arbitrary relations required exponential time in the size of the input and the output. Hence, our results are important for making full disjunction a practical tool in real-world applications.

Future work should address the issue of optimizing the evaluation of queries, under the different semantics that are presented in this work. Semistructured databases, in general, and XML documents, in particular, could be large. Thus, evaluation algorithms should exploit the disk and should not perform the evaluation while the entire database is in memory. Note that for full disjunctions, this problem has received an initial solution in the Separate-and-Join method, which was described in Chapter 6.

Future work should also address the problem of ranking the answers of queries. For example, a user may expect complete answers to be ranked higher than incomplete answers.

It seems that among the four combined semantics that were presented in this work, the flexible-weak semantics is the most useful, due to two reasons. First, query evaluation under this semantics is polynomial, in the size of the input and the output, for any arbitrary query and database. Secondly, it seems that this semantics captures the intended meaning of users better than the other combined semantics. However, additional work is required in order to provide means for measuring the quality of different semantics. A quality measure could assist in determining whether indeed the flexible-weak semantics is better than the other semantics.
Bibliography


Appendix A

Hypergraphs and Acyclic Hypergraphs

There are many equivalent definitions for the notation of “acyclic” hypergraph. We present in this appendix the definitions that we use in the thesis. The definitions that are presented here are generally used by database researchers and are taken from [55, 60].

A hypergraph is a pair $\mathcal{H} = (V, E)$, where $V$ is a set of nodes and $E$ is a set of subsets of $V$. Each subset in $E$ is called an hyperedge of $H$.

Alpha-Acyclic Hypergraphs

Let $E$ and $F$ be two hyperedges in an hypergraph $\mathcal{H}$. Suppose that the attributes in $E - F$ are unique to $E$; that is they appear in no hyperedge other than $E$. Then we call $E$ an ear. The removal of $E$ from $\mathcal{H}$ is called an ear removal. If $E$ intersects no other hyperedge of $\mathcal{H}$, it is also considered an ear and its removal is an ear removal.

In a $GYO$-reduction of a hypergraph, ear removal is applied until there are no ears to remove. A hypergraph is $\alpha$-acyclic if its $GYO$-reduction is the empty hypergraph. Otherwise, it is not $\alpha$-acyclic.

Gamma-Acyclic Hypergraphs

We now introduce $\gamma$-acyclic hypergraphs. The definition is based of [31]. In [31], there is also a notion of $\beta$-acyclic hypergraphs that we do not present here, since $\beta$-acyclic hypergraphs are not discussed in our work.
**Pure Cycle** A *pure cycle* in a hypergraph is a collection of three or more hyperedges and nodes in the form of a cycle as shown in Figure 1. In the pure cycle, each pair of hyperedges $X_i$ and $X_{i+1}$ (and also $X_n$ and $X_1$) has at least one node $A_i$ (or $A_n$ in the case of $X_n$ and $X_1$) in common. Moreover, none of the shared nodes $A_j$ appears in more than two hyperedges of the pure cycle shown by Figure 1. These nodes, however, may appear in hyperedges that are not part of the pure cycle. In the pure cycle, there may be more than one node in the intersection of two adjacent hyperedges, but these nodes must not appear in any other hyperedges of the pure cycle.

**Gamma-Three-Cycle** A *γ-3-cycle* is a set of three hyperedges in the configuration of Figure 2. That is, the nodes $A$, $B$ and $C$ of the hypergraph must exists. Any other regions of the diagram of Figure 2 may or may not be empty, and the regions containing $A$, $B$ and $C$ may also contain other nodes. In other words, there must be some node in all three hyperedges, one that is only in $X$ and $Z$, and one that is only in $Y$ and $Z$. For example, the smallest $γ$-3-cycle consists of the three hyperedges \{ABC, AB, BC\}.

**Gamma-Cycle** A  *γ-cycle* is a cycle of at least three edges, in the form of Figure 1. However, unlike a pure cycle, a $γ$-cycle permits $A_n$ to appear in any of hyperedges $X_2, \ldots X_{n-1}$, as well as $X_1$ and $X_n$. All other nodes in the intersections appear only

---

*Figure 1: A pure cycle in a hypergraph*
in two adjacent $X_i$'s.

**Gamma-Acyclic Hypergraphs**  A hypergraph is $\gamma$-acyclic if and only if it has no $\gamma$-cycle. Equivalently, a hypergraph is $\gamma$-acyclic if and only if it contains no pure cycle and no $\gamma$-3-cycle. A hypergraph is $\gamma$-cyclic if and only if it is not $\gamma$-acyclic.
## Appendix B

The tables that are constructed during the transformation of the movie database (Figure 2.1) to a relation, according to the full-disjunction approach shown in Example 6.4.

### $r_1$

<table>
<thead>
<tr>
<th>Movie</th>
<th>Actor</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>21</td>
<td>Mark Hamill</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
<td>Harrison Ford</td>
</tr>
<tr>
<td>12</td>
<td>25</td>
<td>Natalie Portman</td>
</tr>
</tbody>
</table>

### $r_2$

<table>
<thead>
<tr>
<th>Movie</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Star Wars</td>
</tr>
<tr>
<td>12</td>
<td>León</td>
</tr>
<tr>
<td>13</td>
<td>Magnolia</td>
</tr>
</tbody>
</table>

### $r_3$

<table>
<thead>
<tr>
<th>Movie</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1977</td>
</tr>
</tbody>
</table>

### $r_4$

<table>
<thead>
<tr>
<th>Movie</th>
<th>Director</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>41</td>
<td>George Lucas</td>
</tr>
</tbody>
</table>

### $r_5$

<table>
<thead>
<tr>
<th>Actor</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Kyle MacLachlan</td>
</tr>
<tr>
<td>r6</td>
<td>Actor</td>
</tr>
<tr>
<td>----</td>
<td>-------</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r7</th>
<th>Actor</th>
<th>Filmography</th>
<th>Movie</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
<td>29</td>
<td>Dune</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r8</th>
<th>Actor</th>
<th>Filmography</th>
<th>Movie</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
<td>29</td>
<td>1984</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r9</th>
<th>Actor</th>
<th>Filmography</th>
<th>Movie</th>
<th>Director</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>15</td>
<td>29</td>
<td>42</td>
<td>David Lynch</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

The Proof of Theorem 3.21

We present Algorithm SFPT that evaluates semiflexible path queries w.r.t. tree databases in $O(|Q||D| + |M|)$ runtime, where $|Q|$ is the size of the query, $|D|$ is the size of the database and $|M|$ is the size of the result.

Intuitively, SFPT computes the matchings as follows. A traversal is performed over the database. The correspondence sets are computed during the traversal. Matchings are computed, from the correspondence sets, each time that the traversal reaches a leaf.

We now describe Algorithm SFPT. First we describe three data structured that SFPT uses. How the data structured are being used will be described later. Consider a path query $Q$ and a tree database $D$. The first data structure that SFPT uses is an array $A$ of correspondence sets. In $A$ there is a correspondence set for each variable of $Q$. Initially, all the correspondence sets in $A$ are empty.

The second data structure that SFPT holds is an array $L$ of positive numbers. The array $L$ is used for counting the number of labels in $D$ on paths from the root to a leaf. In $L$ there is an entry for each label $l$ of $Q$. The entry that corresponds to $l$ is denoted $L(l)$. Initially, all the entries in $L$ are equal to 0.

The third data structure in SFPT is a set of matchings $M_T$. The set $M_T$ contains, at each step of the algorithm, the matchings that where computed up to this step. Initially, $M_T$ is empty.

SFPT performs a traversal over $D$ that is similar to a DFS. The traversal is performed using the recursive procedure visit, that is described in Figure 3. The traversal starts by calling visit with the root of the database.
Procedure \textit{visit}(o)

\textbf{Input} a database object $o$;

$D$ is the tree database

$\mathcal{M}_T$ is a set of matching that is being constructed during the traversal;

$L$ is an array of numbers that count the number of instances

for each label on the path to $o$;

$A$ is an array of correspondence sets that is being updated during the traversal;

\textbf{if} $o$ is not the root of $D$ \textbf{then}

\hspace{1em} let $l$ be the label that enters $o$ in $D$;

\hspace{1em} update $L$ by adding 1 to $L(l)$;

\hspace{1em} update $A$ by adding $o$ to all the correspondence sets

\hspace{1em} of variables with incoming label $l$ in $Q$;

\textbf{else} update $A$ by adding $o$ to the correspondence set of the root;

\textbf{if} $o$ is a leaf of $D$ \textbf{then}

\hspace{1em} if for each label $l$ in $Q$, the number of variables in $Q$ with

\hspace{1em} incoming label $l$ is not greater than $L(l)$ \textbf{then}

\hspace{1em} compute $\mathcal{M}$ according to Equation 3.1;

\hspace{1em} add $\mathcal{M}$ to $\mathcal{M}_T$;

\textbf{for each} child $d$ of $o$

\hspace{1em} \textit{visit}(d);

\textbf{if} $o$ is not the root of $D$ \textbf{then}

\hspace{1em} update $L$ by subtracting 1 from $L(l)$;

\hspace{1em} update $A$ by removing $o$ from all the correspondence sets

\hspace{1em} of variables with incoming label $l$ in $Q$;

\textbf{Figure 3:} The recursive procedure \textit{visit} of Algorithm \textit{SFPT}. Algorithm \textit{SFPT} computes the semiflexible matchings of a path query w.r.t. a tree database.

In the traversal there are steps of descending from a node to a child and steps of ascending from a node to the parent. In each step, the arrays $A$ and $L$ are modified according to the node that is visited. Consider a node $o$ in $D$ with incoming label $l$. 

152
When descending to o, L(l) is increased by 1. In A, o is added to the correspondence sets of variables that have incoming label l in Q. When ascending from o, L(l) is decreased by 1. In A, o is removed from the correspondence sets of variables with incoming label l.

When reaching a leaf of D, the algorithm computes the set of matchings w.r.t. the path to this leaf. The computed matchings are added to \( M_T \). The matchings are computed using the correspondence sets of A.

Computing the matchings is as follows. First, the algorithm checks that there are “enough nodes for each label” for at least one matching. This is done by checking that for each label l in Q, if there are k variables in Q with incoming label l then \( L(l) \geq k \). If this test fails, then no matching is added to \( M_T \) and the computation w.r.t. the current leaf is completed. Else, the set of matchings on the left side of Equation 3.1 is computed and added to \( M_T \).

Consider a path query Q and a tree database D. Algorithm SFPT computes the semiflexible matchings of Q w.r.t. D in \( O(|Q||D| + |M|) \) runtime, where |Q|, |D| and |M| are the sizes of the query, the database and the result, respectively.

Proof of Theorem 3.21. We show that SFPT returns the semiflexible matchings of Q w.r.t. D. First we claim that the visit procedure computes the correspondence sets correctly. Consider a call to visit with an object o that is not the root. Let \( \pi \) be the path in the database from the root to the parent of o. Then, for each label l in the query, \( L(l) \) holds the number of objects in \( \pi \) with incoming label l. For each variable \( v \) in Q, the correspondence set of \( v \) in A holds all the objects on \( \pi \) that have an incoming label l, where l is the label on the edge that enters v. The correspondence set of the root holds merely the database root. If o is the root then \( L \) and A hold the initial values. The claim can be proved by a simple induction on the number of calls to visit.

According to the above claim, every matching of Q w.r.t. D is a matching of Q w.r.t. a path of D from the root to a leaf. Essentially, Algorithm SFPT evaluates Q w.r.t. each path in D, from the root to a leaf, and performs a union on the result of the computations w.r.t. the different paths. Correctness of the computation w.r.t. the different paths follows from Proposition 3.20.
Figure 4: An illustration of how to construct a query and a database, according to Lemma 3.22, for the formula \( \varphi = (p_1 \lor p_2 \lor \neg p_3) \land (\neg p_2 \lor \neg p_3 \lor p_4). \) The clauses of \( \varphi \) are denoted by C-1 and C-2, respectively. For each node, the truth-assignment path is the path that comes from the left and the false-assignment path is the path that comes from the right.

We analyze the time complexity of the algorithm. In the algorithm, every node of \( D \) is visited once. The updating of \( L \) and \( A \) in each visit of a node can be done in \( O(1) \). Computing the matchings has the following complexity. Testing that there are “enough object for each labels”, for creating at least one matching, is in \( O(|Q|) \). There can be \( O(|D|) \) leafs for which this test should be performed. Finally, creating the matchings is in \( O(|M|) \) runtime. Thus, the algorithm has \( O(|Q||D| + |M|) \) runtime. \( \square \)

The Proof of Lemma 3.22

Proof of Lemma 3.22. Let \( \varphi = c_1 \land c_2 \land \ldots \land c_m \) be a 3CNF formula and \( U = \{p_1, \ldots, p_n\} \) be the propositional letters appearing in \( \varphi \). We construct from \( \varphi \) a path query \( Q \) and a dag database \( D \) in such a way that a semiflexible matching of \( Q \) w.r.t. \( D \) simulates a satisfying assignment for the propositional letters in \( U \).

First, we explain how to construct \( Q \). The query \( Q \) is a path made of \( m \) edges
such that the \(i\)-th edge in \(Q\) is labeled with \(C^i\). Thus, the query has the form \(v_0, C^i, v_1, \ldots, v_{m-1}, C^j, v_m\).

Secondly, we explain how the database is constructed. The database, has \(n\) nodes denoted by \(o_{p_1}, \ldots, o_{p_n}\). These \(n\) nodes represent the propositional letters in \(U\). In addition, the database has one more node that serves as the root of the database and is denoted by \(r_D\). There are other nodes in the database that we do not name explicitly.

Generally, there are exactly two paths in \(D\) from the root to the node \(o_{p_i}\). Similarly, there are exactly two paths from each node \(o_{p_i}\) to the node \(o_{p_{i+1}}\) (for \(1 \leq i \leq m - 1\)). Next, we describe these pairs of paths.

Given a propositional letter \(p_i\), let the set \(C_T = \{c_{i_1}, \ldots, c_{i_k}\}\) be the set of clauses satisfied by assignment of true to \(p_i\). Let the set \(C_F = \{c_{j_1}, \ldots, c_{j_l}\}\) be the set of clauses satisfied by an assignment of false to \(p_i\). We create a path from \(o_{p_{i-1}} (r_D\) if \(i = 1\)\) to \(o_{p_i}\) that we call the truth-assignment path of \(o_{p_i}\), and a similar path from \(o_{p_{i-1}} (r_D\) if \(i = 1\)\) to \(o_{p_i}\) that we call the false-assignment path of \(o_{p_i}\).

The truth-assignment path of \(o_{p_i}\) is a simple path of \(k + 1\) edges that are labeled with the names of the clauses in \(C_T\) and an additional edge labeled with \(P\), that is, \(o_{p_{i-1}}, C^{i_1}, o_{i_1}, \ldots, o_{i_{k-1}}, C^{i_k}, o_{k}, P, o_{p_i}\). Except the first and the last nodes, all the nodes in the truth-assignment path are fresh, i.e., they are not among the nodes \(r_D, o_{p_1}, \ldots, o_{p_n}\) and they do not appear in any other truth-assignment path or false-assignment path. The last edge is labeled with \(P\) and its role is to make sure that there is a truth-assignment path to \(o_{p_i}\) even if an assignment of true to \(p_i\) does not satisfy any clause. Similarly, the false-assignment path of \(o_{p_i}\) is a simple path of \(k' + 1\) edges that are labeled with the names of the clauses in \(C_F\) and an additional edge that is labeled with \(P\), that is, \(o_{p_{i-1}}, C^{j_1}, o'_{i_1}, \ldots, o'_{i_{k'-1}}, C^{j_{k'}}, o'_{k'}, P, o_{p_i}\). Except the first and the last nodes, all the nodes in the false-assignment path are fresh. Note that even if one or both of the sets \(C_T\) and \(C_F\) are empty, there are still two paths that enter the node \(p_i\) due to the “neutral” edge labeled with \(P\).

There are no nodes and edges in the database in addition to the ones that were described above. Thus, the created database \(D\) is a dag.

We now show that there is a semiflexible matching of \(Q\) w.r.t. \(D\) if and only if
there is a satisfying assignment of $\varphi$.

Assume that $\alpha$ is an assignment to the propositional letters of $U$ that satisfies $\varphi$. We look at a path $\phi$ in $D$ that starts at $r_D$ and goes through the nodes $o_{p_1}, \ldots, o_{p_n}$ as follows. If $\alpha$ assigns true (false) to $p_i$ then $\phi$ goes through the truth-assignment path (false-assignment path) of $o_{p_i}$.

The assignment $\alpha$ satisfies every clause in $\varphi$. Thus, a clause $c_i$ must have a proposition $p_j$ in it such that either $p_j$ is positive in $c_i$ and $\alpha$ assigns true to $p_j$ or $p_j$ is negative in $c_i$ and $\alpha$ assigns false to $p_j$. In the first case, $c_i$ is in $C_T$ of $p_j$, the path $\phi$ goes through the truth-assignment path of $o_{p_j}$, and hence, $\phi$ goes through an edge labeled with the name of $c_i$. In the second case, $c_i$ is in $C_F$ of $p_j$, the path $\phi$ goes through the false-assignment path of $o_{p_j}$, and hence, $\phi$ goes through an edge labeled with the name of $c_i$. Thus, in both cases, the path $\phi$ goes through an edge labeled with the name of $c_i$, and this is true for all the clauses of $\varphi$.

The path $\phi$ is a path in the database, from the root to the only leaf, that goes through at least $m$ edges that are labeled with the names of the $m$ clauses of $\varphi$. A permutation of $\phi$ can transfer these $m$ edges to the beginning of the path and this creates a prefix that can be matched with the query $Q$. Consequently, it shows the existence of a semiflexible matching of $Q$ to $D$.

Conversely, assume that there exists a semiflexible matching $\mu$ of $Q$ to $D$. The existence of $\mu$ entails the existence of a path $\phi$ in $D$ that starts on the root, such that $\phi$ has a permutation with a prefix that matches $Q$. Since $Q$ has edges that are labeled with all the names of the clauses of $\varphi$, it means that $\phi$ must also have edges that are labeled with all the names of clauses of $\varphi$.

We create an assignment $\alpha$ to the propositions of $U$,

$$
\alpha(p_i) = \begin{cases} 
\text{true} & \text{if } \phi \text{ goes through the truth-assignment path of } o_{p_i} \\
\text{false} & \text{if } \phi \text{ goes through the false-assignment path of } o_{p_i}
\end{cases}
$$

We now show that the assignment $\alpha$ satisfies the formula $\varphi$. First we show that if the path $\phi$ goes through an edge labeled with the name of the clause $c_j$, then $\alpha$ satisfies the clause $c_j$. Let $e$ be an edge, on the path $\phi$, labeled with the name of $c_j$.
The edge $e$ is either an edge in a truth-assignment path of some node $o_{p_i}$ or an edge in a false-assignment path of some node $o_{p_i}$.

In the first case, the path $\phi$ goes through a truth-assignment path of the node $o_{p_i}$, so $\alpha$ assigns true to $p_i$. The edge $e$ is in the truth-assignment path of $o_{p_i}$, and this means that the clause $c_j$ is in the set $C_T$ of $p_i$. The set $C_T$ of $p_i$ includes clauses that are satisfied by assigning true to $p_i$, and hence, $\alpha$ satisfies $c_i$. The second case, where the path $\phi$ goes through a false-assignment path of the node $o_{p_i}$, is similar.

For each clause $c_j$ of $\varphi$, the path $\phi$ goes through an edge labeled with the name of $c_j$. This means that $\alpha$ satisfies all the clauses of $\varphi$, and by this satisfies $\varphi$ itself. Thus, the formula $\varphi$ has a satisfying assignment.

The query $Q$ and the database $D$ that were created from $\varphi$ have a polynomial size in $\varphi$. The number of edges in the query is the number of clauses in $\varphi$. In the database, there are the following edges. First there is an edge for each appearance of a propositional letter in a clause of $\varphi$. This sums to $3 \cdot m$ edges. In addition, there are two edges with the label $P$ for each propositional letter of $U$. That is, there are $2 \cdot n$ edges with the label $P$. Thus, for a formula $\varphi$ with $m$ clauses over a set of $n$ propositional letters, there are $3 \cdot m + 2 \cdot n$ edges in the database $D$. We conclude that the construction of the query and the database is polynomial in the size of the formula. 

**The Proof of Lemma 3.57**

*Proof of Lemma 3.57.* Let $\varphi$ be a formula in 3CNF, let $P = \{p_1, \ldots, p_n\}$ be the set of propositional letters appearing in $\varphi$, and let $C = \{C_1, \ldots, C_m\}$ be the set of clauses in $\varphi$. We construct a database $D$ and a query $Q$ in a way such that evaluating the query over the database simulates the assignment of truth values to the propositional letters, and such that flexible matchings are in a one-to-one correspondence with satisfying assignments.

We construct $D$ as follows. Let $r_D$ be the root of $D$. For each clause $C_i \in C$, there is an edge labeled `clause` from $r_D$ to a unique object $ocl_i$. The object $ocl_i$ represents the $i$-th clause of the formula $\varphi$. For each propositional letter $p_j \in P$,
there is an object $o_{true_j}$ and an object $o_{false_j}$. The objects $o_{true_j}$ and $o_{false_j}$ represent the truth values that can be assigned to the propositional letter $p_j$. From each object $ocl_i$ there are seven outgoing edges labeled by $truth-ass_i$ to seven different objects $oass_{i1}, \ldots, oass_{i7}$. The objects $oass_{ik}$ represent the seven possible satisfying assignments for the clause $C_i$. Let $p_{l1}, p_{l2}, p_{l3}$ be the propositional letters that appear in the literals of the clause $C_i$. For each assignment to $p_{l1}, p_{l2}, p_{l3}$ that satisfies $C_i$, we look at the object $oass_{ik}$ that represents that assignment. If in this assignment $p_{lm}$ is assigned true then there is an edge $ass-val_{lm}$ to $o_{true_{lm}}$ in $D$. If in the assignment $p_{lm}$ is assigned false then there is an edge $ass-val_{lm}$ to $o_{false_{lm}}$ in $D$. There are no other objects and edges in $D$. 

Figure 5: The construction of Lemma 3.57 for the formula $\varphi = (p_1 \lor p_2 \lor p_3) \land (\neg p_1 \lor p_2 \lor p_4)$. 

158
We construct a dag-query $Q$. The construction is as follows. The root of $Q$ is $r_Q$. For each clause $C_i \in C$, there is an edge labeled $\text{clause}_i$ from $r_Q$ to a unique variable $vcl_i$. The variable $vcl_i$ represents the $i$-th clause of the formula $\varphi$. For each clause $C_i$, there is a unique variable $vass_i$ in $Q$, and there is an edge, from $vcl_i$ to $vass_i$, labeled by $\text{truth-ass}_i$. For each propositional letter $p_j \in P$, there is a variable $vprop_j$ in $Q$. Let $p_{i_1}, p_{i_2}, p_{i_3}$ be the propositional letters that appear in the literals of the clause $C_i$. There is an edge labeled $\text{ass-val}_{i_m}$ in $Q$ from $vass_i$ to the variable $vprop_{i_m}$, for $m = 1, 2, 3$. There are no other variables and edges in $Q$. From the construction it is easy to see that $Q$ is a dag.

In a flexible matching of $Q$ to $D$, each variable $vprop_i$ is is mapped to either the database object $o\text{true}_i$ or the object $o\text{false}_i$. That corresponds to an assignment of truth values to the propositional letters in $P$. A flexible matching of $Q$ to $D$ is an assignment that satisfies all $vass_i, \text{ass-val}_j, vprop_j$ constraints, by assigning to $vass_i$ and $vprop_j$ database nodes that are indirectly connected by a path in $D$ and with appropriate incoming labels. Hence, an object $oass_{i_k}$ is assigned to each variable $vass_i$ where $oass_{i_k}$ stands for a satisfying assignment for the clause $C_i$. That means that the matching determine directly an assignment that satisfies each clause of $\varphi$, and thus satisfies $\varphi$ itself.

For the other direction, we assume that there is a satisfying assignment to $\varphi$. Let this assignment be $\mu$. The satisfying assignment determine a matching in that an object $o\text{true}_i$ or $o\text{false}_i$ is assigned to the variable $vprop_i$ according to the truth value which is assigned to the propositional $p_i$ by $\mu$. The restriction of $\mu$ to the variables of a clause $C_i$ is a satisfying assignment for $C_i$, and hence one of the seven possible satisfying assignments for $C_i$. That determine which object $oass_{i_k}$ is assigned to $vass_i$. Finally, we get from $\mu$ a flexible matching of $Q$ to $D$.

We conclude that there is a one-to-one correspondence between satisfying truth assignments for $\varphi$ and flexible matchings of $Q$ to $D$.  \qed

159
Database

Figure 6: A database according to the construction of Lemma 5.8 for the formula $\varphi = (p_1 \lor p_2 \lor p_3) \land \neg p_1 \lor p_2 \lor p_4$.

The Proof of Lemma 5.8

Proof of Lemma 5.8. Let $\varphi$ be a formula in 3CNF, let $P = \{p_1, \ldots, p_n\}$ be the set of propositional letters in $\varphi$, and let $C = \{C_1, \ldots, C_m\}$ be the set of clauses of $\varphi$. We construct a query $Q$ and a database $D$ in a way such that if a maximal semiflexible-or-matching maps all the variables of the query to non-null values, then this mapping indicates the existence of a satisfying assignment for $\varphi$. Without loss of generality, we assume the no clause of $\varphi$ contains both a literal and its negation. Obviously, if such a clause exists, it can be removed from $\varphi$ without affecting satisfaction.

We first describe how to construct the database $D$. In $D$ there are four levels of objects below the root. The first level represents the propositional letters of $\varphi$. That is, there are $n$ objects $o_{p_1}, \ldots, o_{p_n}$ and for each object $o_{p_j}$ there is an edge with the
Figure 7: A query according to the construction of Lemma 5.8 for the formula \( \varphi = (p_1 \lor p_2 \lor p_3) \land (\neg p_1 \lor p_2 \lor p_4) \).

Label \( P_j \) from the root to \( o_{P_j} \). The second level consists of \( n \) objects \( o_{R_1}, \ldots, o_{R_n} \).

For each \( 1 \leq j \leq n \), there is an edge with the label \( R_j \) from \( o_{P_j} \) to \( o_{R_j} \). On the third level, for each proposition \( p_j \), there are two objects \( o_{jT} \) and \( o_{jF} \). There is an edge with the label \( P_{jT} \) from \( o_{R_j} \) to \( o_{jT} \) and there is an edge with the label \( P_{jF} \) from \( o_{R_j} \) to \( o_{jF} \). On the fourth level, there are \( m \) nodes \( o_{c_1}, \ldots, o_{c_m} \) that represent the \( m \) clauses of \( \varphi \). There is an edge from \( o_{jT} \) to \( o_{c_i} \) if assigning true to \( p_j \) satisfies the clause \( C_i \), i.e., \( p_j \) is a positive literal in \( C_i \). There is an edge from \( o_{jF} \) to \( o_{c_i} \) if assigning false to \( p_j \) satisfies \( C_i \), i.e., \( C_i \) includes \( \neg p_j \). There are no additional edges or nodes in \( D \).

We now describe how to construct the query \( Q \). As in the database, the query also has four levels of objects below the root. The first level represents the propositional letters of \( \varphi \) and comprises \( n \) variables \( v_{P_1}, \ldots, v_{P_n} \). Each variable \( v_{P_j} \) is connected to the root by an edge with the label \( P_j \). In the second level, there is a pair of variables.
\( v_{jT} \) and \( v_{jF} \) for each propositional letter \( p_j \). From the node \( v_{P_j} \), there is an edge with the label \( P_{jT} \) to \( v_{jT} \) and there is an edge with the label \( P_{jF} \) to \( v_{jF} \). On the third level, there are \( n \) nodes \( v_{R_1}, \ldots, v_{R_n} \). From each node \( v_{jT} \) and from each node \( v_{jF} \) there is an edge with the label \( R_j \) to \( v_{R_j} \). Finally, on the fourth level there are \( m \) variables \( v_{c_1}, \ldots, v_{c_m} \) that represent the \( m \) clauses of \( \varphi \). There is an edge from \( v_{R_j} \) to \( v_{c_i} \) if the propositional letter \( p_j \) appears in the clause \( C_i \), either positively or negatively. There are no additional edges or nodes in \( Q \).

Next, we show that if there exists a semiflexible-or-matching \( \mu \) of \( Q \) w.r.t. \( D \), such that \( \mu \) map all the variable of \( Q \) to non-null, then \( \varphi \) has a satisfying assignment.

Suppose that \( \mu \) is a semiflexible-or-matching of \( Q \) w.r.t. \( D \), such that \( \mu \) does not map any variable of \( Q \) to null. Since \( \mu \) is a semiflexible-or-matching, there exists a rooted fragment \( F \) of \( Q \) such that \( F \) consists of all the nodes of \( Q \) and \( \mu \) is a semiflexible matching of \( F \) w.r.t. \( D \).

**Claim 1** If there is an edge in \( F \) from \( v_{R_i} \) to \( v_{c_k} \), then \( F \) has exactly one of the two edges that enters \( v_{R_i} \), i.e., in \( F \) there is exactly one edge with the label \( R_i \).

As a first step towards proving the claim we provide the next observation. Consider the two edges that are described next. First, an edge from \( o_{iF} \) to the node \( o_{c_k} \). Second, an edge from \( o_{iT} \) to the node \( o_{c_k} \). Only one of this edges can be in \( D \). This is because the edge from \( o_{iT} \) to \( o_{c_k} \) is in \( D \) if the propositional letter \( p_i \) appears positively in \( c_k \). The edge from \( o_{iF} \) to \( o_{c_k} \) is in \( D \) if the propositional letter \( p_i \) appears negatively in \( c_k \). The claim of the observation follows from the assumption that a propositional letter cannot appear both positively and negatively in a clause.

According to the construction of \( D \) and the observation above, for each \( 1 \leq i \leq n \) and \( 1 \leq k \leq m \), only one of the objects \( o_{iF} \) and \( o_{iT} \) has a path to \( o_{c_k} \).

Now, we continue proving the claim. It is easy to see that according to the labels on the edges of both the query and the database, \( \mu \) maps the variables \( v_{R_i}, v_{c_k}, v_{iF} \) and \( v_{iT} \) to the objects \( o_{R_i}, o_{c_k}, o_{iF}, \) and \( o_{iT} \), respectively.

Next, we assume that \( F \) has an edge from \( v_{R_i} \) to \( v_{c_k} \) and in addition has the two edges that enter \( v_{R_i} \)—an edge from \( v_{iF} \) to \( v_{R_i} \) and an edge from \( v_{iT} \) to \( v_{R_i} \). We show that this assumption leads to a contradiction.
According to the above assumption, \( F \) has a path from \( v_{iF} \) to \( v_{c_k} \) and a path from \( v_{iT} \) to \( v_{c_k} \). Since \( \mu \) is a semiflexible matching w.r.t. \( F \), \( \mu(v_{iF}) \) (i.e., \( o_{iF} \)) and \( \mu(v_{c_k}) \) (i.e., \( o_{c_k} \)) should be connected by a path. Similarly, \( \mu(v_{iT}) \) (i.e., \( o_{iT} \)) and \( \mu(v_{c_k}) \) (i.e., \( o_{c_k} \)) should be connected by a path. This contradicts the above observation that only one of the objects \( o_{iF} \) and \( o_{iT} \) has a path to \( o_{c_k} \).

Next, we show how to construct an assignment \( f \) to the propositional letters of \( P \), such that \( f \) satisfies \( \varphi \). The assignment \( f \) is defined as follows. Consider \( 1 \leq i \leq n \).

- If there is a path in \( F \) from a node \( v_{iT} \) to a node \( v_{c_k} \), for some \( 1 \leq k \leq m \),
  then \( f \) assigns true to \( p_i \).

- If there is a path from the node \( v_{iF} \) to a node \( v_{c_k} \), for some \( 1 \leq k \leq m \), then
  \( f \) assigns false to \( p_i \).

- Otherwise, \( f \) maps \( p_i \) to true or to false arbitrarily.

We argue that \( f \) is well defined. This is because there cannot be in \( F \) both a path from the node \( v_{iT} \) to a node \( v_{c_k} \) and a path from the node \( v_{iF} \) to a node \( v_{c_k} \). Otherwise, \( F \) would include two edges that enters \( v_{R_i} \), one from \( v_{iF} \) and the other from \( v_{iT} \). This is a contradiction to Claim .1.

Finally, we show that \( f \) is a satisfying assignment of \( \varphi \). In \( F \), there is a path from the root to each variable. In particular, there is a path to all the variables \( v_{c_1}, \ldots, v_{c_m} \). Consider a path \( \pi \), in \( F \), from the root to \( v_{c_k} \). According to the construction of \( Q \), there is \( 1 \leq i \leq n \) such that \( \pi \) goes either through \( v_{iT} \) or through \( v_{iF} \).

If \( p_i \) goes through \( v_{iT} \) (to \( v_{c_k} \)), then there is a path (actually an edge), in \( D \), from \( o_{iT} \) to \( o_{c_k} \). This is because \( \mu \) is a semiflexible matching of \( F \) and it satisfies the SF-Condition (Definition 3.3) w.r.t. each path of \( F \). According to the construction of \( D \), there is an edge from \( o_{iT} \) to \( o_{c_k} \) if assigning \( p_i \) to true satisfies \( c_k \). According to the definition of \( f \), \( p_i \) is assigned true. Thus, \( f \) satisfies \( c_k \). The case where \( p_i \) goes through \( v_{iF} \) is similar.
For the other direction, let \( f \) be a satisfying assignment for the propositional letters of \( \varphi \). We construct a fragment \( F \) of \( Q \) as follows. The fragment \( F \) contains all the edges that emanate from the root and all the edges that emanate from the variables \( v_1, \ldots, v_n \). For each propositional letter \( p_j \), if \( f \) assigns true to \( p_j \) then \( F \) has an edge from \( v_{jF} \) to \( v_{R_j} \). Otherwise, \( F \) has an edge from \( v_{jF} \) to \( v_{R_j} \). There is an edge from \( v_{R_j} \) to \( v_{c_i} \) if the assignment of \( f \) to \( p_j \) satisfies \( C_j \), i.e., \( f \) assigns true (false) to \( p_j \) and \( p_j \) appears positively (negatively) in \( C_j \). There are no other edges in \( F \).

According to the construction of \( F \) and because \( f \) satisfies all the clauses of \( \varphi \), \( F \) is rooted. We examine the assignment \( \mu \) of \( F \) w.r.t. \( D \), where \( \mu \) is defined as follows. The variables \( v_{P_1}, \ldots, v_{P_n} \) are mapped to the objects \( o_{P_1}, \ldots, o_{P_n} \), respectively. Each variable \( v_{jT} \) is mapped to the object \( o_{jT} \) and, similarly, each variable \( v_{jF} \) is mapped to the object \( o_{jF} \). The variables \( v_{R_1}, \ldots, v_{R_n} \) are mapped to the objects \( o_{R_1}, \ldots, o_{R_n} \), respectively. Finally, each variable \( v_{c_i} \) is mapped to the object \( o_{c_i} \).

It is easy to see that \( \mu \) satisfies the SF-condition w.r.t. paths of \( Q \) that do not reach a node \( v_{c_i} \). For a path \( \phi \) that does reach a node \( v_{c_i} \), note that \( \phi \) goes through a node \( v_{jT} \) only if \( p_j \) appears positively in \( C_i \). In this case, \( o_{jT} \) is connected to \( o_{c_i} \). The case where \( \phi \) goes through \( v_{jF} \) is similar. Thus, \( \mu \) satisfies the SF-condition w.r.t. each path of \( F \) and, therefore, \( \mu \) is a semiflexible matching of \( F \) w.r.t. \( D \).

We show that \( \mu \) is the only maximal semiflexible-OR-matching of \( Q \) w.r.t. \( D \). Consider a fragment \( F' \) of \( Q \) and a semiflexible matching \( \mu' \) of \( F' \) w.r.t. \( D \). It is easy to see that according to the labels on the edges of both the query and the database, for every variable \( v \) in \( F' \), \( \mu'(v) = \mu(v) \). Thus, either \( \mu \) is equal to \( \mu' \) or \( \mu \) subsumes \( \mu' \). Thus, any semiflexible-OR-matching of \( Q \) w.r.t. \( D \) that is different from \( \mu \) is not maximal.

\[ \square \]

**The Proof of Proposition 6.7**

**Proof of Proposition 6.7.** We start with the following claim.

**Claim.** There is an attribute \( A \) in \( R_j \) such that \( A \) is not an attribute of any schema of \( G_1 \) (\( G_2 \)) other than \( R_j \).
We show that the claim is true. Let $R_i \ (i \neq j)$ be a schema in $G_2$ that has a non-empty intersection with $R_j$. Such schema exists because $G_2$ is connected. If the claim was not true, then all the attributes of $R_j$ were in schemas of $G_1$, including the attributes of the intersection of $R_i$ and $R_j$. In this case, $R_i$ was connected to a node of $G_1$ and to a node of $G_2$. This is a contradiction to the assumption that $G_1$ and $G_2$ are a separation of $G$ w.r.t. $R_j$, and it proves the claim.

We now continue with the proof of the proposition. Suppose that $t$ is a tuple of $FD(G)$. We show that $t$ is also a tuple of $FD(G_1) \cong FD(G_2)$.

The tuple $t$ is a maximal integrated tuple. Thus, it is generated from $m \ (m \leq n)$ tuples $t_{i_1}, \ldots, t_{i_m}$ of the relations $r_{i_1}, \ldots, r_{i_m}$, respectively. Without loss of generality, let $R_{i_1}, \ldots, R_{i_{k-1}}$ be nodes of $G_1$, $R_{i_{k+1}}, \ldots, R_{i_m}$ be nodes of $G_2$ and $R_{i_k}$ be the node that separates $G_1$ and $G_2$, i.e., the only node that is in both $G_1$ and $G_2$. Note that each one of the sets $\{R_{i_1}, \ldots, R_{i_k}\}$ and $\{R_{i_k}, \ldots, R_{i_m}\}$ could be empty. This happens when the non-null portion of $T$ consists of attribute of schemas of $G_1$ and does not include attributes of schemas of $G_2$, or vice-versa.

Consider the following three assertions, for the case where $\{R_{i_1}, \ldots, R_{i_k}\}$ is not empty. First, the tuples $t_{i_1}, \ldots, t_{i_k}$ are join consistent because they are part of the integrated tuple $t$. Secondly, the schemas $R_{i_1}, \ldots, R_{i_k}$ are connected. This is because $R_{i_1}, \ldots, R_{i_m}$ are connected (otherwise, $t$ would not be in $FD(G)$) and none of the schemas $R_{i_1}, \ldots, R_{i_{k-1}}$ has an edge to one of the schemas $R_{i_{k+1}}, \ldots, R_{i_m}$. Thirdly, there are no $R_{i_l}$ in $G_1$ and a tuple $t_l$ of $r_l$ such that (1) $t_l$ is join consistent with $t_{i_1}, \ldots, t_{i_k}$, (2) $t_l$ has an attribute $A$ such that $A$ does not belong to any of the schemas $R_{i_1}, \ldots, R_{i_k}$, and (3) $R_l$ is connected to one of the relations $R_{i_1}, \ldots, R_{i_k}$. Otherwise, $t$ would not be maximal and would be subsumed by the join of $t$ and $t_l$.

The above three assertions show that the universal tuple $t_1$ whose generators are $t_{i_1}, \ldots, t_{i_k}$, is a maximal integrated tuple w.r.t. the schema nodes of $G_1$ and the corresponding relations of these schemas. Similarly, it can be shown that if $\{R_{i_k}, \ldots, R_{i_m}\}$ is not empty then the universal tuple $t_2$ whose generators are $t_{i_k}, \ldots, t_{i_m}$, is a maximal integrated tuple w.r.t. the schema nodes of $G_2$ and the corresponding relations of these schemas.
Thus, if \( \{R_{i_1}, \ldots, R_{i_k}\} \) is not empty, then \( t_1 \) is an element of \( FD(G_1) \). Otherwise we consider \( t_1 \) as a null tuple. If \( \{R_{i_k}, \ldots, R_{i_m}\} \) is not empty, then \( t_2 \) is an element of \( FD(G_2) \). Otherwise we consider \( t_2 \) as a null tuple.

Next, we show that the join of \( t_1 \) and \( t_2 \) is in \( FD(G_1) \bowtie FD(G_2) \). Suppose that both \( t_1 \) and \( t_2 \) are non-null. Consider the generators of \( t_1 \) and the generators of \( t_2 \). Since all these tuples are generators of \( t \), they are pairwise join consistent. Thus, \( t_1 \) and \( t_2 \) are join consistent.

The attributes in the intersection of the schemas of \( t_1 \) and \( t_2 \) are the attributes of \( R_{i_k} \). We claim that both tuples do not have a null in any attribute of \( R_{i_k} \) and \( t_1[R_{i_k}] = t_2[R_{i_k}] \). The claim is true because the tuple \( t_{i_k} \) is a generator of both \( t_1 \) and \( t_2 \). The reason that the tuple \( t_{i_k} \) is a generator of both \( t_1 \) and \( t_2 \) is the following. The generators of \( t \) belong to relations with schemas that form a connected graph. However, the only node that connects schemas of \( G_1 \) with schemas of \( G_2 \) is \( R_{i_k} \). Thus, if both \( t_1 \) and \( t_2 \) are non-null then their join is in \( FD(G_1) \bowtie FD(G_2) \).

Consider the case where \( t_2 \) is null. In this case, the set of generators of \( t_1 \) does not include \( t_{i_k} \), or else, \( t_{i_k} \) would have been a generator of \( t_2 \) and \( t_2 \) would not be null. We argue that there is an attribute \( A \) in \( R_{i_k} \) such that \( t_1[A] \) is null. Let \( A \) be an attribute that is belong to \( R_{i_k} \) and does not belong to any of the schemas of \( G_1 \). There is such an attribute according to Claim .2. Since \( t_{i_k} \) is not a generator of \( t_1 \), \( t_1[A] \) is null.

Because \( t_1[A] \) is null for some attribute \( A \) in \( R_{i_k} \), in the outerjoin of \( FD(G_1) \bowtie FD(G_2) \), \( t_1 \) cannot be joined to any tuple of \( FD(G_2) \). Thus, \( t_1 \) appears in the result padded with null values.

The case where \( t_1 \) is null is similar to the case where \( t_2 \) is null. Thus, in all three cases: when \( t_1 \) is null, when \( t_2 \) is null and when neither of the tuples is null, \( t \) is in \( FD(G_1) \bowtie FD(G_2) \).

For the other direction, suppose that \( t \) is a tuple in \( FD(G_1) \bowtie FD(G_2) \). We show that \( t \) is in \( FD(G) \).

There are three cases to consider. First, \( t \) is created by a join of two tuples \( t_1 \) and \( t_2 \) of \( FD(G_1) \) and \( FD(G_2) \), respectively. Second, \( t \) is created from a tuple \( t_1 \) of
Consider the first case, i.e., \( t \) is created by a join of two tuples \( t_1 \) and \( t_2 \) of \( FD(G_1) \) and \( FD(G_2) \), respectively. We show that the generators of \( t_1 \) and \( t_2 \) create a maximal integrated tuple w.r.t. \( r_1, \ldots, r_n \).

First, we show that the generators of \( t_1 \) and \( t_2 \) are join consistent. Let \( X_1 \) be the generators of \( t_1 \) and \( X_2 \) be the generators of \( t_2 \). The tuples of \( X_1 \) (\( X_2 \)) are join consistent and connected because \( t_1 \) (\( t_2 \)) is in \( FD(G_1) \) (\( FD(G_2) \)). Let \( X \) be the union of \( X_1 \) and \( X_2 \). Because \( t_1 \) and \( t_2 \) are joined in an outerjoin operation, each two tuples, one from \( X_1 \) and one from \( X_2 \), are join consistent. Thus, the tuples in \( X \) are join consistent.

Secondly, we show that the generators of \( t_1 \) and \( t_2 \) are connected. Consider \( R_j \)—the articulation node that the separation is w.r.t. it (the schema that was denoted \( R_{ik} \) in the first part of the proof). The tuples \( t_1 \) and \( t_2 \) do not have a null on any of the attributes of \( R_j \). Otherwise, \( t_1 \) and \( t_2 \) were not joined in the outerjoin. Because of Claim 2, there are no null values in \( t_1[R_j] \) only if there is a tuple \( t' \) in \( r_j \) that is a generator of \( t_1 \). Due to the same argument and because \( t_1 \) and \( t_2 \) are equal on the attributes of \( R_j \), \( t' \) is also a generator of \( t_2 \). The tuple \( t' \) connects the generators of \( t_1 \) and the generators of \( t_2 \). Hence, the tuples of \( X \) are connected.

Thirdly, the join of the generators of \( t_1 \) and \( t_2 \) is a maximal integrated tuple. If the join was not maximal, then either \( t_1 \) was not maximal, w.r.t. the relations whose schemas are nodes of \( G_1 \), or \( t_2 \) was not maximal, w.r.t. the relations whose schemas are nodes of \( G_2 \). The above three arguments show that \( t \) is a maximal integrated tuple and it is in \( FD(G) \).

Consider the second case, i.e., \( t \) is created from a tuple \( t_1 \) of \( FD(G_1) \) by padding \( t_1 \) with nulls. Let \( X \) be the set of generators of \( t_1 \). Obviously, the tuples in \( X \) are join consistent and connected. Thus, they are the generators of an integrated tuple. This integrated tuple is maximal w.r.t. the relations whose schemas are in \( G_1 \). We need to show that this integrated tuple is also maximal w.r.t. the relations \( r_1, \ldots, r_n \).

We claim that the generators of \( t_1 \) do not include a tuple of \( r_j \) (Recall that the separation of \( G \) is w.r.t. \( R_j \)). We assume that the claim is incorrect and show that
this leads to a contradiction.

Assume that there is a generator \( t_j \) of \( t_1 \) which is a tuple of \( r_j \). The schema \( R_j \)
is a node of \( G_2 \). According to the definition of the full disjunction, there is a tuple
\( t_2 \) in \( FD(G_2) \) such that \( t_j \) is one of its generators (note that it could be that \( t_j \) is the
only generator of \( t_2 \)). The tuple \( t_2 \) is join consistent with \( t_1 \) because \( t_1 \) and \( t_2 \) have
the same values for the attributes of \( R_j \). Hence, the join of \( t_1 \) and \( t_2 \) is a tuple in the
outerjoin of \( FD(G_1) \) and \( FD(G_2) \). That yield a contradiction to the assumption that
there is no tuple in \( FD(G_2) \) which is joined with \( t_1 \). This shows that the generators
of \( t_1 \) do not include a tuple of \( r_j \).

Consider a tuple over the schema nodes of \( G \) whose generators are the generators
of \( t_1 \). This tuple is equal to \( t \), that is, it is the result of padding \( t_1 \) with null values.
Since the generators of \( t_1 \) do not include a tuple of \( r_j \), it follows from Claim \( \cdot \) \( .2 \)
that there is an attribute \( A \) in \( R_j \) such that \( t_1[A] \) is null. This shows that any set of
generators that includes the generators of \( t_1 \) and a tuple of a relation whose schema
is in \( G_2 \) is not connected and join consistent. This shows that \( t \) is maximal.

Thus, the tuple \( t \) is a maximal integrated tuple and, hence, it is in \( FD(G) \). The
third case is similar to the second case. Therefore, in all three cases, \( t \) is in \( FD(G) \)
and the proof is completed.

\[ \Box \]

The Proof of Theorem 6.10

Before we give the proof of Theorem 6.10, we provide some background on the compu-
tation of the full disjunction, without a projection, for relations with connected,
\( \gamma \)-acyclic schemas.

Consider a a set of relation schemas that form a connected, \( \gamma \)-acyclic hypergraph.
The algorithm SOJO receives these schemas and returns a sound outerjoin ordering
of the schemas. A sound outerjoin ordering is an expression that consists of the
given schemas and outerjoin operations. In the expression, each schema appears
exactly once. Furthermore, if in this expression, the schemas are replaced with their
respective relations and the expression is computed, the result of the computation
is the full disjunction of the relations. The exact definition of SOJO and the
proof of its correctness are given in [55].

In Sketch, SOJO performs the following computation steps. In the first step, a Bachman diagram $\mathcal{B}$ is constructed from the given schemas. In subsequent steps, SOJO is applied recursively to $\mathcal{B}$. Recall that the nodes of a Bachman diagram are either schemas or attribute sets that are intersection of schemas. A node is minimal if it is not contained in any other node of the diagram.

The recursion is as follows. If $\mathcal{B}$ comprises a single node $R$ then $R$ is returned. Else, a minimal node $Z$ is chosen among the nodes of $\mathcal{B}$. According to $Z$, $\mathcal{B}$ is decomposed into two sub-diagrams $\mathcal{B}_1$ and $\mathcal{B}_2$. The decomposition has the following property. The set $Z$ is the intersection of the following two sets: the set of attributes in the nodes of $\mathcal{B}_1$ and the set of attributes in the nodes of $\mathcal{B}_2$. SOJO is executed on $\mathcal{B}_1$ and on $\mathcal{B}_2$. Suppose that $E_1$ and $E_2$ are the expressions that SOJO returns when recursively called with $\mathcal{B}_1$ and $\mathcal{B}_2$. Then $E_1 \circ E_2$ is returned for $\mathcal{B}$.

Next, we explain how to modify SOJO in order to solve the projection problem. Consider a set of relations $r_1, \ldots, r_n$ with schemas $R_1, \ldots, R_n$, respectively. Let $X$ be a given set of attributes. Our goal is to compute the projection on $X$ of the full disjunction of $r_1, \ldots, r_n$. To do so, we modify SOJO as follows. First, on each recursive call of the algorithm, the input includes a set of projection attributes, in addition to a diagram. Secondly, instead of returning an expression that comprises merely the schemas and outerjoins, the returned expression also includes projections.

The modified recursion is as follows. Let $\mathcal{B}$ be the Bachman diagram and $X$ be the projection attributes that are given in the input. For the case where $\mathcal{B}$ comprises a single node $R$, the projection of $R$ on $X$ is returned.\footnote{Suppose that $R$ is a schema and $X$ is a set of attributes that is not contained in $R$. We assume, in this case, that the projection of $R$ on $X$ is equal to the projection of $R$ on $X \cap R$.} For the case where $\mathcal{B}$ has more than one node, $\mathcal{B}$ is decomposed into two sub-diagrams $\mathcal{B}_1$ and $\mathcal{B}_2$, according to a minimal node $Z$. The algorithm is called recursively—once with $\mathcal{B}_1$ and then with $\mathcal{B}_2$. In both cases, the projection attributes are $X \cup Z$. Suppose that $E_1$ and $E_2$ are the expressions that the recursive calls produce. Then, the projection of $E_1 \circ E_2$ on $X$ is returned.

SOJO is defined for connected schemas, It is easy to see that when the schema
graph is not connected, SOJO can be applied to each connected component. The results for the connected components are combined using outerunion. In an outerunion of two relations \(r_1\) and \(r_2\) with schemas \(R_1\) and \(R_2\), respectively, the schema of the outerunion is the union of \(R_1\) and \(R_2\). The outerunion itself is a relation that consists of all the tuples of \(r_1\) and \(r_2\), where a tuple of one relation is padded with null values in the attributes of the other relation.

We can now prove Theorem 6.10.

**Proof of Theorem 6.10.** We prove the theorem by first showing that the modified SOJO correctly computes the projection on \(X\) of the full disjunction. Later we will show that the runtime of the modified SOJO is polynomial in the size of the input and output.

Suppose that \(r_1, \ldots, r_n\) are relations with schemas \(R_1, \ldots, R_n\), respectively. Let \(X\) be a given set of projection attributes. We denote by \(F_X\) the projection of the full disjunction of \(r_1, \ldots, r_n\) on \(X\). We denote with \(F_X^{\text{sojo}}\) the result of evaluating the expression that the modified SOJO returns for \(R_1, \ldots, R_n\) and \(X\). It should be shown that \(F_X = F_X^{\text{sojo}}\).

First, we want to show that attributes that are needed for the join are not removed too early because of the projections. To see this, consider a pair of schemas \(R_i\) and \(R_j\). Let \(E_1\) and \(E_2\) be two expressions that the algorithm computes in one of the recursive calls, such that \(R_i\) appears in \(E_1\) and \(R_2\) appears in \(E_2\). \((E_1\) and \(E_2\) are unique for the given schemas, because a schema cannot appear more than once in the final expression). We define the schema of an expression to be the schema of the relation that is the result of computing the expression after replacing relation schemas with their corresponding relations.

**Claim 3** Both the schema of \(E_1\) and the schema of \(E_2\) contain the shared attributes of \(R_i\) and \(R_j\), i.e., contain \(R_i \cap R_j\).

To see why Claim 3 is true, consider the decomposition that returns the expression \(E_1 \bowtie^o E_2\). Let \(Z\) be the node of the diagram that this decomposition is with respect to it. Since \(Z\) is a node in a Bachman diagram, the set \(Z\) is the intersection
of the following two sets. The set of attributes in schemas that appear in $E_1$ and
the set of attributes in schemas that appear in $E_2$. Thus, $Z$ contains $R_i \cap R_j$. Now,
one should notice that at each recursive call, the set of projection attributes is only
increased. So, all the projections, in sub-expressions of $E_1$ and $E_2$, are on sets that
contain $Z$, and, consequently, these sets contain $R_i \cap R_j$.

The next claim shows that attributes of $X$ are not discarded by projections.

**Claim 4** In each expression $E$ that is produced during the run of the modified SOJO
algorithm, $E$ is the projection of some sub-expression of $E$ on a set of attributes that
contains $X$.

Again, this claim follows from the fact that, in the recursive calls, the set of
projection attributes is always increased, i.e., no attribute is removed.

Next, we prove correctness by showing containment in both directions. Suppose
that $t_x$ is a tuple in $\mathcal{F}_X^{\to}$. It follows from Claim 3 that the tuples that were joined
to produce $t_x$ are join consistent. Thus, in $\mathcal{F}_X$ there is a tuple that is equal to $t_x$ or
subsumes $t_x$.

Before showing containment in the other direction, we provide a notation. Con-
sider an expression $E$ than contains the schemas $R_1, \ldots, R_n$. The evaluation result
of $E$ is the relation that is produced by computing $E$ after replacing each schema $R_i$
with its corresponding relation $r_i$. We denote the evaluation result of $E$ as $\text{eval}(E)$.

Now, we show containment in the other direction. Suppose that $t_x$ is a tuple in
$\mathcal{F}_X$. Then there is a tuple $t$ in the full disjunction of $r_1, \ldots, r_n$, such that $t_x$ is the
projection of $t$ on $X$. Let $t_{i_1}, \ldots, t_{i_m}$ be the generators of $t$.

Consider the expression $E^u$ that the unmodified SOJO produces for $R_1, \ldots, R_n$.
Then $t$ is in $\text{eval}(E^u)$. In the evaluation of $E^u$, $t$ is produced by a sequence of joins
w.r.t. the tuples $t_{i_1}, \ldots, t_{i_m}$. We show that in the modified SOJO there is a similar
sequence of joins, w.r.t. projections of $t_{i_1}, \ldots, t_{i_m}$, and this sequence produces $t_x$ or
a tuple that subsumes $t_x$.

Consider the expression $E^m$ that the modified SOJO produces. The two expres-
sions $E^u$ and $E^m$ have the same order of outerjoin operations. The difference between
them is the additional projections in $E^m$. Given a sub-expression $E_1$ of $E^n$, we denote as $\text{Proj}(E_1)$ the sub-expression of $E^m$ that contains exactly the same schemas as $E_1$. Such sub-expression exists because the decomposition to sub-expressions in SOJO is the same as the decomposition in the modified SOJO.

Consider the evaluation of $E^m$ after replacing each schema $R_i$ with its corresponding relation $r_i$. The following claim holds,

**Claim 5** Suppose that $E'$ is a sub-expression of $E^n$. Let $t'$ be a tuple in $\text{eval}(E')$, such that all the generators of $t'$ are among $t_{i_1}, \ldots, t_{i_m}$. Then there is a set of attributes $X'$ that contains $X$, such that $t'[X']$ is in $\text{eval}(\text{Proj}(E'))$.

We prove Claim 5 by induction on the structure of $E'$. If $E'$ is a relation schema $R_{ij}$, for some $1 \leq j \leq m$, then $\text{Proj}(E')$ is a projection of $R_{ij}$ on a set $X'$. According to Claim 4, $X'$ contains $X$. In this case, $t_{ij}$ is in $\text{eval}(E')$ and $t_{ij}[X']$ is in $\text{eval}(\text{Proj}(E'))$. Thus, the claim holds.

Suppose that $E'$ has the form $E_1 \circ E_2$. Let $t'$ be a tuple in $\text{eval}(E')$, such that the generators of $t'$ are among $t_{i_1}, \ldots, t_{i_m}$. Then one of the following three cases must hold. First, $t'$ is in the projection of $\text{eval}(E_1)$ on a set of attributes $X'$. Second, $t'$ is in the projection of $\text{eval}(E_2)$ on a set of attributes $X'$. Third, $t'$ is a projection on a join of a tuple of $\text{eval}(E_1)$ and a tuple of $\text{eval}(E_2)$. In the first two cases, according to Claim 4, $X'$ contains $X$ and the claim holds.

We consider the third case. There are tuples $t_1$ and $t_2$ such that (1) $t_1$ and $t_2$ are in $\text{eval}(E_1)$ and $\text{eval}(E_2)$, respectively, (2) the generators of $t_1$ and $t_2$ are among $t_{i_1}, \ldots, t_{i_m}$, and (3) $t'$ is the join of $t_1$ and $t_2$.

By the induction hypothesis, there are sets of attributes $X_1$ and $X_2$ that contain $X$, such that $t_1[X_1]$ is in $\text{eval}(\text{Proj}(E_1))$ and $t_2[X_2]$ is in $\text{eval}(\text{Proj}(E_2))$. Since $t_1$ and $t_2$ are join consistent, $t_1[X_1]$ and $t_2[X_2]$ are join consistent. Hence, there is an $X'$ such that $\text{eval}(\text{Proj}(E'))$ is a projection of the join of $t_1[X_1]$ and $t_2[X_2]$ on $X'$. According to Claim 4, $X'$ contains $X$. This proves Claim 5.

From Claim 5 follows that there is a set of attributes $X'$ that contain $X$ such that $\text{eval}(E^m)$ contains $t'[X']$. The tuples $t'[X']$ and $t_x$ are created from the same generators. Hence, $t'[X']$ is equal to $t_x$ or subsumes $t_x$. Hence, there is a tuple in
\( \mathcal{F}^{\text{SOJO}}_X \) that is either equal to \( t_x \) or subsumes \( t_x \). This proves the other side of the containment.

So far, we showed the correctness of the algorithm. Next, we show that computing a projection of a full disjunction, using an expression that the modified SOJO produces, can be done in polynomial time in the size of the input and output.

First, note the following three simple fact. (1) The expression that modified SOJO produces does not include any schema more than once. (2) Computing a projection of a relation is in polynomial time in the size of the relation. (3) Computing an outerjoin of a pair of relations is in polynomial time in the size of the two relations. Thus, we only need to show that each intermediate relation has a bounded size w.r.t. the size of the input and the final result.

Suppose that \( E \) is the expression that the modified SOJO produces for \( R_1, \ldots, R_n \) and \( X \). Let \( E_1 \) be a sub expression of \( E \) that is returned from one of the recursive calls. In addition, let \( \mathcal{F}_X \) be the result of evaluating \( E \) and \( \mathcal{F}_1 \) be the result of evaluating \( E_1 \).

**Claim 6** The number of tuples in \( \mathcal{F}_1 \) is less than or equal to \( s_r \times f \), where \( s_r \) is the number of tuples in all the relation \( r_1, \ldots, r_n \) and \( f \) is the size of the result, i.e., the number of tuples in \( \mathcal{F}_X \).

To prove Claim 6 we consider two cases. In the first case, \( E_1 \) is a projection on an input relation. Obviously, in this case, \( \mathcal{F}_1 \) has at most \( s_r \) tuples. Thus, the claim holds.

The second case is when \( E_1 \) is a projection of \( E'_1 \bowtie E'_2 \) on \( X \cup Z \), where \( E'_1 \) and \( E'_2 \) are sub-expressions. The set \( Z \) is a node in the Bachman diagram. We show that there cannot be too many tuples in \( E_1 \).

Every tuple \( t \) is \( \mathcal{F}_1 \) has two parts (not necessarily disjoint). One part is the projection of \( t \) on \( X \). The other part is the projection of \( t \) on \( Z \). The projection of \( t \) on \( X \) is part of a tuple of \( \mathcal{F}_X \). Thus, there are at most \( f \) tuples in the projection of \( \mathcal{F}_1 \) on \( X \).
According to the way decomposition is performed in the algorithm, \( Z \) is a minimal node in the Bachman diagram. Hence, for each schema \( R_j \) that appears in \( E_1 \), the set \( Z \) is either contained in \( R_j \) or disjoint from \( R_j \). Thus, the projection of \( t \) on \( Z \) is a part of a tuple in one of the relations that appear in \( E_1 \). In other words, the projection of \( t \) on \( Z \) cannot be created by a Cartesian product or a join of two tuples from two different relations. Otherwise, the schemas of these relations do not contain \( Z \) and are not disjoint from \( Z \), which is a contradiction to the minimality of \( Z \). Hence, the number of tuples in the projection of \( F_1 \) on \( Z \) is at most \( s_r \).

Let \( C \) be the Cartesian product of the following two relations: the projection of \( F_1 \) on \( Z \) and the projection of \( F_1 \) on \( X \). The number of tuples in \( F_1 \) is less than the number of tuples in \( C \). Also, the number of tuples in \( C \) is at most \( s_r \times f \). This proves Claim 6 and proves the theorem. \( \Box \)