DATA MINING TECHNIQUES FOR STRUCTURED AND SEMISTRUCTURED DATA

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Svetlozar Nestorov

July 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Jeffrey Ullman
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Rajeev Motwani

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Shalom Tsur

Approved for the University Committee on Graduate Studies:

Abstract

Data mining is the application of sophisticated analysis to large amounts of data in order to discover new knowledge in the form of patterns, trends, and associations. With the advent of the World Wide Web, the amount of data stored and accessible electronically has grown tremendously and the process of knowledge discovery (data mining) from this data has become very important for the business and scientific-research communities alike.

This doctoral thesis introduces Query Flocks, a general framework over relational data that enables the declarative formulation, systematic optimization, and efficient processing of a large class of mining queries. In Query Flocks, each mining problem is expressed as a datalog query with parameters and a filter condition. In the optimization phase, a query flock is transformed into a sequence of simpler queries that can be executed efficiently. As a proof of concept, Query Flocks have been integrated with a conventional database system and the thesis reports on the architectural issues and performance results.

While the Query-Flock framework is well suited for relational data, it has limited use for semistructured data, i.e., nested data with implicit and/or irregular structure, e.g. web pages. The lack of an explicit fixed schema makes semistructured data easy to generate or extract but hard to browse and query. This thesis presents methods for structure discovery in semistructured data that alleviate this problem. The discovered structure can be of varying precision and complexity. The thesis introduces an algorithm for deriving a schema-by-example and an algorithm for extracting an approximate schema in the form of a datalog program.

Acknowledgements

First and foremost, I would like to thank my advisor professor Jeff Ullman. Throughout my years at Stanford and beyond, he constantly has provided me with support, challenges, and inspiration. He has helped me in innumerable ways, but most importantly, he helped me make the difficult transition from being a mathematician to being a computer scientist.

I am very grateful to professor Rajeev Motwani and Dr. Dick Tsur for their continued guidance and for serving on my oral and reading committees.

I would also like to thank Serge Abiteboul for the numerous discussions on topics ranging from semistructured data to Eric Cantona and "*Allez les bleus!*"

As a member of the fabulous Database Group at Stanford, I had the privilege to work and learn with professor Hector Garcia-Molina and professor Jennifer Widom on the TSIMMIS and Lore projects, respectively.

The countless hours spent in Gates were made (almost) fun by my fellow students Wilburt, Shiva, Vassilis, Arturo, Jason, Yue, Ramana, Roy, Jan, and all other DB-ers. Special thanks go to my fellow data miners (current Googlers) Sergey, Larry, and Craig for the wild Midas meetings. I would also like to thank Marianne and Andy, who keep the DB group running smoothly.

Finally, I would like to thank my parents, Atanaska and Evtim, for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The amount of data stored and available electronically has been growing at an ever increasing rate for the last decade. In the business community, companies collect all sorts of information about the business process such as financial, payroll, and customer data. The data is often among the most valuable assets of a business. In the scientific community, a single experiment can produce terabytes of data. Subsequently, there is growing demand for methods and tools that analyze large volumes of data. However, even storing, let alone analyzing, such huge amounts of data presents many new obstacles and challenges. An oft used metaphor that “we are drowning in data, and yet starving for knowledge” sums up the situation perfectly. The field of data mining has emerged out of this necessity.

Data mining is broadly defined as the process of finding “patterns” from large amounts of data. The definition is necessarily vague because it has to encompass the vast array of methods, techniques, and algorithms from various fields such as databases, machine learning, and statistics. To obfuscate things even further, data mining is often considered to be only a step, or be it the most important one, in the knowledge discovery process. The knowledge discovery process involves several other pre-mining and post-mining steps such as data cleansing and data visualization. In the present thesis, the focus is on data mining from the database perspective.

Initially, the exclusive target of data mining was well-structured data, such as relational data. The canonical example is the discovery of correlations among goods purchased at a large supermarket or a department store. The literature often cites the slightly amusing discovery that people who buy diapers are likely to buy beer. Data mining of structured data has also been successfully applied to various scientific disciplines ranging from astronomy to genomics.

The emergence of the World Wide Web (WWW) and the integration of various heterogeneous data source has introduced large amounts of data with less-than rigid structure. The database community has adopted the term *semistructured data* for such data. By definition semistructured data has some implicit regularity but there is no explicit schema to which that data conforms. Later in this thesis we argue that even XML data can be considered semistructured.

Our contributions, presented in this thesis, are twofold. First we present a unifying framework for expressing, optimizing, and processing of data mining problems over relational data. Second, we define the problem of mining structure from semistructured data and present two different solutions that discover structural information of varying degrees of precision and complexity.

1.1 Data Mining for Structured Data

The state of the art in data mining for structured data is many different algorithms that operate on limited types of data. Furthermore, most data-mining methods are at best loosely-coupled with relational DBMS, thus not taking advantage of the existing database technology. In this thesis, we propose a framework, called *query flocks*, that allows the declarative formulation of a large class of data-mining queries over relational data. We also present a method for systematic optimization and efficient processing, called *query flock plans*, of such queries. A distinctive feature of the query-flock framework is that it can be integrated in a tightly-coupled manner with relational DBMS. We show that, by using query flock plans, we can utilize fully the query processing capabilities of relational databases without sacrificing performance.

1.2 Data Mining for Semistructured Data

The importance of semistructured data has been recognized in the database community and is emphasized by the flurry of research activities in the last several years. The emergence of XML and its rapid adoption by the e-commerce companies has made semistructured data equally important for the business community. However, since the proliferation of semistructured data has been relatively recent, there is a lack of tools and methods for analysis of such data. Standard data-mining techniques developed for structured data are difficult to apply and have been shown to be ineffective[NAM98]. In this thesis, we make the following two contributions that address the problem of analyzing semistructured data.

First, we define and motivate the problem of discovering structure information from semistructured data. In Chapter 4 we describe a commonly accepted data model for semistructured data based on directed labeled graph. Then we show that having an explicit structure information has great benefits for browsing, querying, and storing of semistructured data.

Second, we present two methods for structure discovery for semistructured data. The first method, presented in Chapter 5, is based on the determinization and minimization of nondeterministic finite automaton. The approach compresses the labeled graphs into the graph with the least number of vertices that has the same set of labeled paths as the original data. Thus, we consider the data to be perfect as the method does not allow for noise. The second method, presented in Chapter 6 is based on datalog programs that describe the data approximately. We introduce a model for the data noise and describe an algorithm for deriving an approximate schema that trades accuracy for conciseness.

1.3 Thesis Organization

The rest of the thesis is organized as follows. The next chapter introduces query flocks and shows that many data mining problems can be expressed in the query flock framework. The problem of processing query flock efficiently is addressed in Chapter 3. This chapter introduces the systematic optimization of query flocks as query flock plans and presents an integration with relational DBMS in a tightly-coupled fashion. Chapter 4 introduces the problem of data ming semistructured data. Chapters 5 and 6 present two different approaches to discovering structure from semistructured data. The thesis ends with Chapter 7 that presents our conclusions.

Chapter 2

Query Flocks

2.1 Introduction

Data mining, from its inception, has targeted primarily relational data. There are numerous reasons that warrant this almost exclusive focus; here, we recount three of the most important ones. Firstly, most of the business data, accumulated by companies for many decades, is well structured and can be thought of as relations. Indeed, almost all of the financial-market information, sales data, payroll data, etc., is in the form of flat tuples with attributes. Secondly, relational data is almost always available electronically. Companies often use relational database management systems (RDBMS) such as Oracle and Informix, to store their data persistently. The database technology developed and deployed in RDBMS is relatively mature. Besides efficient storage and retrieval, this technology provides many additional features such as concurrency control, recoverability, and high availability. Thirdly, the rigid structure of relational data makes it amenable to complex queries and analysis such as on-line analytical processing (OLAP), the predecessor of data mining. Furthermore, the impact of finding interesting patterns and trends in relational data can be very high because the discoveries are often easy to understand and exploit in practice.

There are many different techniques and algorithms for relational data that can be classified as data mining. In the current literature, there are roughly four broad classes: clustering, classification, sequence analysis, and associations. While the first three classes of problems have been studied in machine learning and statistics for many decades, associations, also called association rules, have emerged relatively recently and are widely regarded as the flagship of data mining. Furthermore, the underlying assumption behind clustering and classification is the a-priori existence of a model of which the actual data is just an observed instance. Association rules, on the other hand, are data-centric, and patterns that emerge do not have to be combined to derive a complete model. Furthermore, the amount of data that has been subjected to association-rule mining is several orders of magnitude larger than the amount of data normally used in classification and clustering. In a nutshell, association rules are data mining from a database perspective while classification, clustering, and sequence analysis have a machine-learning bias. In this thesis, we consider data mining for structured data from a database perspective. As a consequence, association rules will be featured more prominently than the other three classes of mining problems.

Query flocks, the subject of this chapter, is an elegant framework for a large class of data mining problems over relational data. The main features of query flocks are:

- Declarative formulation of a large class of mining queries.
- Systematic optimization and processing of such queries.
- Integration with relational DBMS, taking full advantage of existing capabilities.

Declarative formulation: Even within the field of association-rule mining, there are many different techniques and algorithms that work only with limited types of data. For example, many association-rule algorithms assume that the data is given as a single relation with only two attributes. With query flocks, on the other hand, many different kinds of mining problems can be expressed declaratively in a simple logic language. Associations are the most natural example, but one can also express certain kinds of clustering, classification, and sequence analysis.

Systematic optimization and processing: The current state of the art in data mining of structured data is ad-hoc optimization techniques that only apply to specific problems and limited types of data. In query flocks, mining queries are optimized and processed systematically in the form of query flock plans. Query flock plans generalize the a-priori technique for a larger class of mining problems and can also be augmented with other techniques such as hashing and partitioning.

Integration with relational DBMS: Most of the current mining systems are loosely-coupled, at best, with a relational DBMS. Thus, they forgo the opportunity to use most of the existing capabilities of RDBMS. Query flocks, on the other hand, can be easily integrated with a relational database. Furthermore, the integration can be tightly coupled, meaning that the query-processing capabilities of the database are utilized fully.

This chapter is devoted to the query flock framework and will focus mainly on the declarative formulation of mining problems as query flocks. The next chapter will detail the systematic processing and optimization as query flock plans and the tightly-coupled integration with relational DBMS.

2.1.1 Chapter Organization

The rest of this chapter is organized as follows. Section 2.2 reviews the basics of association rules and the a-priori technique. In Section 2.3 we introduce query flocks expressed as parameterized conjunctive queries augmented with arithmetic expressions and negation, and filter conditions. Section 2.4 considers the generalization of the a-priori technique in query flocks and sets the stage to consider, in Chapter 3, the implementation of the generalized a-priori technique as query flock plans. In Section 2.5, we show how different data mining problems, such as classification and sequence analysis can be expressed as query flocks. Section 2.6 concludes this chapter.

2.2 Association Rules

Association-rule mining is widely regarded as the flagship of data mining. Since its introduction in [AIS93], the problem of mining association rules from large databases has been investigated in numerous studies. The topics range from improving the basic *a-priori* algorithm [BMTU97], to mining generalized or multilevel rules [SA95], to parallel algorithms [HKK97] and incremental maintenance [CHNW96]. The vast majority of these studies share the basic *a-priori* technique based on levelwise pruning. Recently, however, there has been some work on finding different kinds of association rules where the basic a-priori technique cannot be applied [MCD$^+$00, FMU00]. In this section, we will focus on "traditional" association rules.

Before we proceed with the formal definition of association rules, we highlight the intuition behind them with the following example. Consider a typical supermarket where every day thousands of shoppers come to the checkout registers with baskets of supermarket items. An observant store manager may note that many customers tend to purchase

Items: $I = \{beer, bread, chips, milk, salsa\}$
Baskets: $B = \{\{bread, milk\}, \{bread, chips, milk\}, \{beer, chips\}$
$\{beer, bread, chips, salsa\}, \{beer, chip, salsa\}, \{beer, bread, milk\}\}$

Figure 2.1: Simple example of market basket data

several specific items together, e.g., bread and milk, beer, chips, and salsa, or vodka and caviar. Furthermore, the store manager may notice that not only many people buy beer, chips, and salsa together, but also people rarely buy just beer and chips. In other words, customers who buy beer and chips are especially likely to buy salsa. We call such a pattern an *association rule*. The goal of association-rule mining is to discover such patterns automatically from large amounts of data.

Knowing the shopping patterns of their customers is extremely valuable to the store managers and supermarket chains. Turning association rules into increased margins and profits is beyond the scope of this thesis. In order to convince the reader, however, of the usefulness of mining association rules, we suggest some simple promotional uses of discovered rules. Consider the rule that many customers who buy beer and chips tend to buy salsa. A creative store manager may decide to put beer, chips, and *premium* salsa on the same isle. Thus, customers who buy beer and chips from this isle will likely purchase the pricey salsa. An alternative approach may be to put frequently purchased items, such as bread and milk, on the opposite ends of the store so that customer will spend more time browsing the isles and thus do more impulsive buying.

As we have seen from the example above, association rules are naturally described in the context of a *market-basket* data. Formally, we define *market-basket data* as follows:

Definition 2.1 *Let* $I = \{i_1, i_2, \cdots, i_k\}$ *be a set of* k *elements, called* items. *Let* $B = \{b_1, b_2, \cdots, b_n\}$ *be a set of* n *subsets of* I. *We call* $b_i \subseteq I$ *a* basket *of items.*

In retail transaction data, the items correspond to individual products for sale, and the baskets correspond to all products bought by a consumer during a particular shopping trip. Figure 2.1 shows a simple example consisting of five items and six transactions of supermarket data.

The abstraction of market baskets is applicable not only to retail data but also to a variety of different domains. For example, consider text data where each item corresponds to a word and each basket corresponds to a text document. Another example is the World Wide Web (WWW), where the items correspond to web pages and baskets correspond to the set of pages linked from a given page.

An association rule is intended to capture the extent of co-occurrence of two sets of items in the given basket data. We say that the itemset P is associated with the itemset Q, and write $P \rightarrow Q$, where $P \subseteq I$ and $Q \subseteq I$. There are several quantities that measure the importance of an association rule. In the original definition [AIS93], we have the following two:

- $support(P \rightarrow Q) = \dfrac{|\{b_i|(P \cup Q) \subseteq b_i\}|}{n}$
- $confidence(P \rightarrow Q) = \dfrac{|\{b_i|(P \cup Q) \subseteq b_i\}|}{|\{b_i|P \subseteq b_i\}|}$

The *support* is the fraction of all n baskets that contain all items from both P and Q. The *confidence* is the fraction of the baskets which contain P that also contain Q. Note that both the confidence and support of an association rule are real numbers in $[0, 1]$. In practice, we only care about association rules that have support and confidence above certain thresholds. These are natural restriction for the supermarket data. Recall the supermarket example and the rule that beer and chips are associated with salsa, written in our formal notation as $\{beer, chips\} \rightarrow \{salsa\}$. For a supermarket situated in a yuppie neighborhood, such as Palo Alto, this rule may not be very useful because the amount of beer and chips sold is very low. Hence, the high-support requirement.

For another concrete example, consider the basket data from Figure 2.1. Suppose that we are interested in association rules with support of at least 0.33, which translates into 2 baskets, and confidence of at least 0.66. Table 2.1 list all such rules.

In this chapter, we will use an alternative definition of the market basket data as a relational database. For simplicity, but without loss of generality, we assume that the market-basket data is given as a relation `baskets(BID,Item)`. This relation consists of pairs of a basket "ID" and an item that appeared in that basket. Table 2.2 shows the `baskets` relation for the example basket data from Figure 2.1.

2.2.1 The A-Priori Optimization

As the example of association rules from Table 2.1 shows, there are numerous possible associations for any basket data. Even if we limit our search to associations of pairs of

Association Rule	Support	Confidence
$\{bread\} \rightarrow \{milk\}$	0.5	0.75
$\{milk\} \rightarrow \{bread\}$	0.5	1.00
$\{beer\} \rightarrow \{chips\}$	0.5	0.75
$\{chips\} \rightarrow \{beer\}$	0.5	0.75
$\{salsa\} \rightarrow \{beer\}$	0.33	1.00
$\{salsa\} \rightarrow \{chips\}$	0.33	1.00
$\{beer, chips\} \rightarrow \{salsa\}$	0.33	0.66
$\{beer, salsa\} \rightarrow \{chips\}$	0.33	1.00
$\{chips, salsa\} \rightarrow \{beer\}$	0.33	1.00

Table 2.1: Association rules with support threshold of 0.33 and confidence threshold of 0.66.

BID	Item
100	bread
100	milk
101	bread
101	chips
101	milk
102	beer
102	chips
103	beer
103	bread
103	chips
103	salsa
104	beer
104	chips
104	salsa
105	beer
105	bread
105	milk

Table 2.2: Example `baskets` relation.

Algorithm 2.1

Input: I – set of k items
B – set of baskets
$minSup$ – support threshold
Output: F_i – sets of frequent itemsets of size $i = 1..k$

```
// Initialization
   C_1 = I
   i = 1
   F_j = ∅ for j = 1..k
// Iterations
   while C_i ≠ ∅ do
      compute support(P) from B for all P ∈ C_i
      F_i = {P|P ∈ C_i, support(P) >= minSup}
      C_{i+1} = generate_candidates(F_i)
      i = i + 1
   end while
return F_j for j = 1..k
```

Figure 2.2: Basic a-priori algorithm.

items, such as $\{bread\} \rightarrow \{milk\}$, there are $\Theta(k^2)$ potential rules, where k is the number of items. There is an important optimization technique, called *a-priori*, that makes the search for itemsets with high support very efficient. The *a-priori* technique, introduced in [AIS93], is one of the main reasons for the apparent success and popularity of association rule mining.

The key idea of a-priori is to use levelwise pruning to reduce the number of itemsets with potentially high support. From the definition of support, it follows immediately, that if an itemset P has high support than *any* subset of P also has high support. Conditions that obey this property are called *monotone* [TUC+98] (or *anti-monotone* in [NLHP98]). The basic a-priori algorithm is shown in Figure 2.2

The procedure *generate_candidates* takes the set of all frequent items of size i, F_i, and returns the set of all *candidate* frequent itemsets of size $i+1$. Itemset P is chosen as a candidate *iff* any subset of P of size i is frequent, i.e., is in F_i.

We illustrate how Algorithm 2.1 works with the example basket data in Figure 2.1 with $minSup = 0.33$.

- In the first iteration, the support of all single items is computed. All of them pass the threshold, so we have $F_1 = C_1 = I$. Then, $C_2 = generate\_candidates(F_1)$ contains all pairs of items in I.

- In the second iteration, the support of all itemsets in C_2 is computed, so we have that $F_2 = \{\{bread, milk\}, \{bread, chips\}, \{beer, chips\}, \{beer, bread\}, \{beer, salsa\}, \{chips, salsa\}\}$. Then, $C_3 = \{\{beer, chips, salsa\}, \{beer, bread, chips\}\}$.
- In the third iteration, we find that $F_3 = \{\{beer, chips, salsa\}\}$. This is the final iteration because $C_4 = \emptyset$.

2.3 Definition of Query Flocks

Before we give the formal definition of a *query flock*, consider our running example of supermarket data. Suppose, we are interested in finding all frequent itemsets of size 2, i.e., pairs of items. In principle, we can enumerate all possible pairs and for each pair $\{X, Y\}$ ask the query "How many baskets contain both X and Y". Then, we can check whether the answer of each query is greater than the given support threshold. If so, we add the pair $\{X, Y\}$ to our final result. If we designate X and Y as parameters, then we have many identical queries except for the values of their parameters. Hence, the idea of a *flock* of queries, or a *query flock*. Thus, a query flock is the parameterized query, that represents all possible simple queries, with instantiated parameters, and the filter condition that we apply to the answer of each simple query.

Formally, we define a query flock as

1. One or more *predicates* that represent the given data stored as relations.
2. A set of *parameters*.
3. A parameterized *query*.
4. A *filter* that specifies conditions that the result of each instantiated query must satisfy in order for a given assignment of values to the parameters to be acceptable.

The meaning of such a query flock is the set of tuples that represent the "acceptable" assignments of values for the parameters. We determine the acceptable parameter assignments by, in principle, trying all such assignments in the query, evaluating the query, and checking whether the result passes the filter conditions. Of course, there are more efficient ways to compute the meaning of a query flock, and these optimizations are the subject of the next chapter.

It is important to distinguish between parameterized queries and query flocks. A query flock is a query about its parameters. The result of the flock is *not* the result of the parameterized query that is used to specify the flock.

2.3.1 Our Languages for Flocks

The notion of a query with a filter condition representing a data mining problem has been proposed first in [Man97]. The key idea is to express both the query and the filter as logic statements. Thus, the filter can be as complex as the query. For example, the filter may state that one of the items in a market basket must be bread. In query flocks, the role of the filter is limited to a condition about the result of the query. To represent the above example filter in query flocks, we would explicitly mention bread in the query part of the query flock. Another proposal for a query with a filter condition is presented in [NLHP98]. In contrast to [Man97], this proposal has very limited query form and a complex filter language involving set variables. However, the proposed query form is nothing more that basic association rules which limits the type of mining problems that can be expressed.

We will use as our query language "conjunctive queries" [CM77], augmented with arithmetic, negation, and union. The filter condition will be expressed in a SQL-like manner over the result of the query. We will use Datalog ([Ull88]) notation to express conjunctive queries. Aside from being the language of choice in the database community, Datalog has two major advantages specific to the query flock framework:

1. The notion of "safe query" for Datalog is directly applicable to query optimizations for query flocks.

2. The generalization of the a-priori technique for query flocks and more complex optimization tricks are most apparent and intuitive when expressed in Datalog.

In order to specify the query part of a query flock, in Datalog terminology ([Ull88]), we need to provide the following:

1. *Extensional* predicates that represent the given data stored as relations.

2. A set of *parameters*, which we will always denote with names beginning with `$`.

3. *Intentional* predicates expressed as conjunctive queries with added arithmetic and negation (or union of such queries) over the extensional predicates.

For the filter language we use SQL conditions similar to the ones in the HAVING clause [UW97]. A condition is an equality or inequality of two expressions. Each expression can involve the following:

1. Aggregate functions: `COUNT, SUM, AVG, MIN, MAX`.
2. Basic arithmetic: $(+, -, *, /)$.
3. Standard mathematical functions such as log, $sqrt$, abs, etc.
4. Constants (real numbers).
5. Attributes (columns) of intentional or extensional predicates.

2.3.2 Market Basket Analysis as a Query Flock

As our first example, we will consider the simplest market-basket problem as a query flock. We are given a relation `baskets(BID,Item)` as the only extensional predicate representing the underlying data. Table 2.2 gives example contents of the `baskets` relation. Recall that market basket analysis is about finding those pairs of items `$1` and `$2` that appear in at least c baskets.

Query Flock 2.1 (Basic Market Baskets)

QUERY:

```
answer(B) :- baskets(B,$1)
        AND baskets(B,$2)
```

FILTER:

```
COUNT(answer.B) >= 20
```

Example 2.1 *Query Flock 2.1 finds pairs of items that appear in at least 20 baskets. For any values of* `$1` *and* `$2`*, the query asks for the set of baskets B in which items* `$1` *and* `$2` *both appear. The answer relation for this pair of items is the set of such baskets. Then, the filter condition requires that the set of such baskets number at least 20. The result of the*

$1	*$2*
beer	diapers
diapers	beer
bread	milk
milk	bread
...	...

Table 2.3: Example result of the market-basket query flock.

query flock is thus the set of pairs of items (`$1`*,* `$2`*) such that there are at least 20 baskets containing both items* `$1` *and* `$2`*. Table 2.3 gives an example of the flock result.*

This query flock easily generalizes to finding sets of k items that appear together for any fixed k. Finding something more complex, however, like the set of maximal sets of items that appear in at least c baskets (regardless of the cardinality of the set of items), is more awkward and would be expressed as a sequence of query flocks with increasing cardinalities, with each flock depending on the result of the previous one.

2.3.3 Multiple Intentional Predicates

The most natural query flocks, and indeed the flocks for which we have the most promising optimization techniques, involve support as the filter condition; Query Flock 2.1 is such a flock. It is possible to represent confidence, interest, and other conditions as filters, using our SQL-like filter language. However, it is necessary to allow the query portion of a flock to produce several relations as its result. Thus, we need multiple intentional predicates so that we can express the filter condition. Furthermore, we can have several different filter conditions, e.g. high support and high confidence.

Query Flock 2.2 (Market Baskets)

QUERY:

```
answer1(B) :- baskets(B,$1)
     AND baskets(B,$2)

answer2(B) :- baskets(B,$1)
```

FILTER:

```
2 * COUNT(answer2.B) >= COUNT(answer1.B) (high confidence)
COUNT(answer.B) >= 20 (high support)
```

Example 2.2 *Suppose we want to find pairs of items* `$1` *and* `$2` *such that, in addition to high support, the confidence of* `$2` *given* `$1` *is at least 50%. The flock can be written as in Query Flock 2.2. There are two intentional predicates, which we call* `answer1` *and* `answer2`*. The first counts the number of baskets containing both items, while the second counts the number of baskets containing the first item. The high-confidence condition asks that the ratio of these counts be at least 1/2. The high-support condition is the same as before.*

2.3.4 Adding Arithmetic, Union, and Negation

The market-basket problem corresponds to a very simple query flock. The query is a conjunctive query with only positive subgoals. In order to express more complex mining problems we need to add more power to our query language. The extensions to conjunctive queries that we will allow are:

1. Negated subgoals.
2. Arithmetic subgoals, e.g., $X < Y$, where X and Y are variables, parameters, or constants.

We will refer to this broader class of conjunctive queries as *extended conjunctive queries*. Note that the expressions allowed in the arithmetic subgoals are somewhat different than the filter language. In particular, there are no aggregate functions in the arithmetic subgoals. In addition, we will allow a query that is the union of these extended CQ's. However, as with the original CQ's, we assume that extended CQ's follow the conventional *set semantics* rather than bag semantics, where duplicate tuples are allowed. Some of our claims would not hold for bag semantics.

As a simple example of where arithmetic subgoals are useful, the original flock for market baskets, Query Flock 2.1, produces each successful pair of items twice, e.g. (bread,milk) and (milk,bread). We can restrict the result to have each pair of items appear only in lexicographic order if we add an arithmetic condition to the query, as:

```
answer(B) :- baskets(B,$1) AND baskets(B,$2) AND $1 < $2
```

The next example illustrates the use of negation in the query part of a query flock.

Example 2.3 *The following is an example of a query flock that searches for unexplained side-effects. That is, we want to find symptoms* `$s` *and medicines* `$m` *such that there are many patients (and as before, we take 20 to be the threshold of "many") that exhibit the symptom and are taking the medicine, yet the patient's disease does not explain the symptom. The underlying data with which we work consists of the following relations:*

1. `diagnoses(Patient, Disease)`*: The patient has been diagnosed with the disease.*
2. `exhibits(Patient, Symptom)`*: The patient exhibits the symptom.*
3. `treatments(Patient, Medicine)`*: The medicine has been prescribed for the patient.*
4. `causes(Disease, Symptom)`*: The disease is known to cause the symptom.*

The query flock for the problem described above appears as Query Flock 2.3. Without loss of generality, we make the assumption that each patient has one disease only. We can handle patients with multiple diseases simultaneously by extend our query-flocks language to allow intermediate predicates. In particular, we need a predicate that relates patients to the set of symptoms caused by any of their diseases.

Query Flock 2.3 (Side Effects)

```
QUERY:
    answer(P) :-
         exhibits(P,$s) AND
         treatments(P,$m) AND
         diagnoses(P,D) AND
         NOT causes(D,$s)
FILTER:
    COUNT(answer.P) >= 20
```

In Query Flock 2.3, the parameterized query asks for the set of patients P that exhibit a symptom `$s`, are receiving medicine `$m`, have disease D, and yet the disease D doesn't explain the symptom `$s`. The filter requires that there be at least 20 patients taking medicine `$m` and exhibiting unexplained symptom `$s`.

The next example illustrates the use of union of conjunctive queries in the query part of a query flock.

Example 2.4 *In this example, we are looking for web sites (*domains*) that are "similar" based on their "neighborhoods", i.e. the pages that they link to and from. Our formal definition of "similar" web domains is based on the count of:*

1. *The number of different domains of pages that have links to pages of both domains.*
2. *The number of different pages that are linked to from pages of both domains.*

The query flock that finds pairs of similar web domains is based on the following extensional predicates:

1. `page(URL, Domain)`*: The page with the given URL resides in the given Domain.*
2. `link(URL1, URL2)`*: There is a hyperlink from the page with URL1 to the page with URL2.*

Query Flock 2.4 shows the flock that finds pairs of web domains (sites) such that there are domains with pages pointing to pages from both domains and pages that are pointed to from pages from both domains. In order to get every pairs of domains only once we require that the first domain, `$1` lexically precede the second domain, `$2`.

As in all of our examples in this chapter, we have taken 20 occurrences as the threshold of significance. Note that the count in the filter is counting answers, which may be either domains or urls. Naturally, we assume that there are no values in common between these two types of identifiers.

Our definition of similar pages may seem somewhat contrived at first but there are good reasons for our choice. First, consider our choice of counting only the unique domains of pages pointing to pairs of web sites. With this choice, we prevent intentional spamming where any individual domain can create many pages pointing to the pair of web sites in order to make them appear similar to the system. Second, consider our choice of using unique pages within the pair of web sites to point to the same page. In this case, both web sites must agree on a page that they link to, so there is no possibility of intentional spamming.

Query Flock 2.4 (Similar Web Domains)

QUERY:

```
answer(D) :-
     page(U,$1) AND
     page(V,$2) AND
     page(W,D) AND
     link(W,U) AND
     link(W,V) AND
     $1 < $2

answer(P) :-
     page(U,$1) AND
     page(V,$2) AND
     page(P,X) AND
     link(U,P) AND
     link(V,P) AND
     $1 < $2
```

FILTER:

```
COUNT(answer.#1) >= 20
```

2.4 Generalization of the A-Priori Technique

Let us start with the basic market-basket flock (Query Flock 2.1). The a-priori technique for this problem translates into finding frequent items first and them using the result to find frequent pairs. In query flocks, we can find all frequent items with the very simple flock shown below:

Query Flock 2.5 (Frequent Items)

QUERY:

```
answer(B) :- baskets(B,$1)
```

FILTER:

```
COUNT(answer.B) >= 20
```

Note that this query flock is simpler than the original flock. In particular, the result of its query part is a superset of the result of the query part of Query Flock 2.1. Furthermore the result of this flock is a superset of the projection on its first attribute of Query Flock 2.1.

Thus, in query flocks, the a-priori technique entails finding simpler query flocks which results are supersets of the result of the original flock. These simpler query flocks, have simpler query part and keep the same filter.

2.4.1 Containment for Conjunctive Queries

For conjunctive queries ([CM77], [Ull89], [AHV95]), there is a straightforward way to find simpler queries. The simplest way for the result of a query Q_1 to be a superset of the result of a query Q_2 is for it to be provable for any database, which we write $Q_1 \supseteq Q_2$. For conjunctive queries, this containment is decidable, using the technique of containment mappings ([CM77]). As a corollary of the containment-mapping theorem we have that the only way $Q_1 \supseteq Q_1$ can hold is if Q_1 is constructed from Q_2 by

1. choosing a subset of the subgoals of Q_2, and
2. changing some variable names into new unique variable names.

Changing the variable names simply decreases the amounts of joins we have to perform but keeps the number of predicates in the query the same. Thus the resulting query is not simpler than the original one. Choosing a subset of the subgoals of Q_2, however, reduces the number of predicates in the query and thus, makes it simpler and less expensive to compute. In both cases, the result of Q_1 is a superset of the result of Q_2. Since changing variable names does not make the query simpler, we will only consider queries Q_1 obtained by choosing a subset of the subgoals of Q_2.

2.4.2 Safe Queries

Not every subset, however, of the subgoals of a conjunctive query forms an "acceptable" query. By acceptable, we mean a query that makes sense by producing a finite result. An infinite result may occur if any of the variables that appear in the head of the query does not appear in any of the subgoals that form the body of the query. Note that a query with an infinite result means that regardless of the filter any values of the parameters will be in the result of the query flocks. Thus, the query flock itself will have an infinite result which is of no use. The condition that guarantees the finite result of conjunctive queries is called *safety*. This condition has been studied before ([Ull88]) as a way to restrict Datalog queries to be equivalent to relational algebra. Conjunctive queries that satisfy the condition are called *safe queries*.

Rule 2.1 *(***Safety Rule for Conjunctive Queries)**
Each variable that appears in the head of the query must also appear in the body of the query.

Example 2.5 *Consider the basic market-basket Query Flock 2.1. Its query part is shown below:*

```
answer(B) :- baskets(B,$1) AND baskets(B,$2)
```

There are only two nontrivial subqueries formed by taking a nonempty, proper subset of the subgoals,

```
answer(B) :- baskets(B,$1)
```

and

```
answer(B) :- baskets(B,$2)
```

Note that the two queries are symmetric with respect to parameters `$1` *and* `$2`*. Furthermore, the original query is also symmetric to itself with respect to the two parameters. Thus, we can use either of the simpler queries to prune the values for the two parameters. Any value that is not in the result of the simpler flock (having the simpler query and the same filter) can be a value for neither* `$1` *nor* `$2` *in the result of the original flock.*

Example 2.6 *Our next example extends Query Flock 2.1 to find frequent pairs of items where at least one of the items is "expensive". The relation* `expensive(Item)` *contains all high-margin items. Association rules involving such items are of a particular interest because increasing their sales results in higher profits.*

Query Flock 2.6 (Expensive Market Baskets)

QUERY:

```
answer(B) :- baskets(B,$1)
     AND baskets(B,$2)
```

```
          AND expensive($1)
FILTER:
     COUNT(answer.B) >= 20
```

There are six nontrivial subqueries of the query part of Query Flock 2.6. Using the safety rule we can eliminate only one of them, namely `answer(B) :- expensive($1)`, since the head variable, B does not appear in the body of the query. Then, we have to choose which of the remaining five subqueries to evaluate which is the subject of the next chapter.

We may summarize the generalization of a-priori for CQ's without negation or arithmetic, as follows:

Rule 2.2 (General Optimization Rule for Conjunctive Queries)
Given a query flock that consists of a query part Q and a support filter, consider all safe subqueries of Q obtained by deleting one or more subgoals. The result of a query flock formed by such a safe subquery and the same filter limits the possible values of the flock's parameters.

2.4.3 Safe Queries with Negation and Arithmetic

When we expand our horizon beyond conjunctive queries to the Datalog queries with negation and arithmetic that we have been using, matters get more complex in several ways. First, the discovery of containing queries is not as easy. There are decision procedures — [Klu82] or [ZO93] for Datalog with arithmetic, and [LS93] for Datalog with negation, including arithmetic. However, there are some cases where the containing query cannot be characterized as a subset of the subgoals of the contained query.

Since these cases are unusual, we propose to continue our restriction that we look only at subsets of the subgoals of the query that defines the query flock. We then have only to augment our search with the generalized notion of what a safe query is. There are now three conditions that must be satisfied ([UW97]):

Rule 2.3 (Safety Rule for Extended Conjunctive Queries)

1. *Every variable that appears in the head must appear in a nonnegated, nonarithmetic subgoal of the body.*

2. *Every variable that appears in a negated subgoal of the body must appear in a non-negated, nonarithmetic subgoal of the body.*

3. *Every variable that appears in an arithmetic subgoal of the body must appear in a nonnegated, nonarithmetic subgoal of the body.*

However, parameters are *variables*, not constants, as far as the above safety conditions are concerned. Since they cannot appear in the head, they are not affected by rule (1). The last two rules apply to parameters as well as to explicit variables.

Example 2.7 *Consider Query Flock 2.3 from Example 2.3. Its query part, shown below, contains a negated subgoal.*

```
answer(P) :-
     exhibits(P,$s) AND
     treatments(P,$m) AND
     diagnoses(P,D) AND
     NOT causes(D,$s)
```

There are 14 nontrivial subqueries and we need to determine which ones are safe. First, to satisfy condition (1) of Rule 2.3, one of the subgoals must include the head variable P. That condition rules out only one possible subquery:

```
answer(P) :- NOT causes(D,$s)
```

Note that this query makes no sense, since it is trying to count a number of patients, but the only information we have says that some disease D does not cause the symptom `$s`.

Condition (2) requires that if we pick the subgoal `NOT causes(D,$s)`*, then since variable D and parameter* `$s` *appear in this subgoal, we must also pick a positive subgoal that has D in it and a positive subgoal that has* `$s` *in it. That is, if we pick* `NOT causes(D,$s)`*, then we must also pick both* `diagnoses(P,D)` *and* `exhibits(P,$s)`*, the only positive subgoals with D and* `$s`*, respectively. Thus, condition (2) again rules out the subquery above that has only* `NOT causes(D,$s)` *in its body and also rules* *out the other five subqueries that have this subgoal but do not have both of* `exhibits(P,$s)` *and* `diagnoses(P,D)`.

Condition (3) is not applicable since there are no arithmetic subgoals.

The remaining eight subqueries are candidates for use in an optimization where we use the subquery to limit the possible values for `$s`, `$m`. *The next chapter details how we choose the queries to use in the optimization. Below we outline the meaning of some likely candidates:*

1. `answer(P) :- exhibits(P,$s).`
   At least 20 patients exhibit the symptom.

2. `answer(P) :- treatments(P,$m).`
   At least 20 patients must have been given the medicine.

3. `answer(P) :- diagnoses(P,D) AND exhibits(P,$s) AND NOT causes(D,$s).`
   There are at least 20 patients with a disease that does not cause a symptom they exhibit.

4. `answer(P) :- exhibits(P,$s) AND treatments(P,$m).`
   There are at least 20 patients taking the medicine and exhibiting the symptom.

2.4.4 Extension to Unions of Datalog Queries

Suppose a query flock consists of a union of Datalog queries of the type that we have been considering. We can construct a query that provides an upper bound on the result of the union if we take the union of queries that provide an upper bound on each query individually. Thus, we must look for a subquery for each query in the union. Each query must be safe, in the sense described in Section 2.4.2. If so, then the size of the result of the union of the subqueries will be a bound on the size of the result for the original query. We may thus use the union of subqueries to eliminate values of a parameter or parameters that cannot possibly appear in the result of the query flock.

Rule 2.4 (General Optimization Rule for Unions of Conjunctive Queries) *Given a query flock that consists of a query part described as a union of conjunctive queries* $Q_1, Q_2, \ldots, Q_n$ *and a support filter, consider all unions of safe subqueries* $P_1, P_2, \ldots, P_n$ *such that* P_i *is formed by deleting one or more subgoals from* Q_i*, for* $i = 1, 2, \ldots, n$. *The* *result of a query flock formed by such a union and the same filter limits the possible values of the flock's parameters.*

Example 2.8 *Consider the union in the query part of Query Flock 2.4 from Example 2.4. Suppose we want to find a subquery that involves only one domain* `$1`*. There are several choices for each of the two queries.*

First consider the query that counts the different domains:

```
answer(D) :- page(U,$1) AND page(V,$2) AND page(W,D) AND
     link(W,U) AND link(W,V) AND $1 < $2
```

Any safe subquery about `$1` *must contain* `page(W,D)` *and* `page(U,$1)`*. In the absence of any statistics of* `page` *and* `link`*, the most promising subquery is shown below. The rationale will be explained in details in the next chapter.*

```
answer(D) :- page(U,$1) AND page(W,D) AND link(W,U)
```

Second, consider the query that counts web pages:

```
answer(P) :- page(U,$1) AND page(V,$2) AND page(P,X) AND
     link(U,P) AND link(V,P) AND $1 < $2
```

Any safe subquery about `$1` *must contain* `page(U,$1)` *and at least of the following three subgoals:* `page(P,X)`, `link(U,P)`, `link(V,P)`*. The most promising safe subquery is shown below:*

```
answer(P) :- page(U,$1) AND link(U,P)
```

Thus, a web site cannot be a candidate for `$1` *unless the sum of the following two quantities is at least 20:*

1. *Number of domains that have at least one page pointing to a page on the given web site.*

2. *Number of pages linked from the given web site.*

2.5 What Can We Express with Query Flocks?

One of the main objective of query flocks is to allow the declarative formulation of a large class of mining queries. Indeed, we have already seen several different examples of mining problems that can be expressed as query flocks. In this section we will show examples of other typical mining problems such as classification, clustering, and sequence analysis phrased as query flocks.

2.5.1 Classification

A typical problem in classification is to find the best k attributes in order to predict accurately the class of certain instances. Here, we consider the following modified problem:

Suppose we have the following data:

- `attributes(InstanceID, AttributeName, AttributeValue)`
- `class(InstanceID, ClassName)`

We want to find all pair of attributes and their corresponding values such that knowing the two values, we can predict the class of an instance, with 80% accuracy (based on the underlying data). The following flock expresses this problem:

Query Flock 2.7 (Classification)

QUERY:
```
answer1(I) :-
     attributes(I,$1,$2) AND
     attributes(I,$3,$4) AND
     class(I,$5)

answer2(I) :-
     attributes(I,$1,$2) AND
     attributes(I,$3,$4)
```

FILTER:
```
COUNT(answer1.I) >= 0.8 * COUNT(answer2.I)
```

The result of Query Flock 2.7 is a set of quintuples `($1,$2,$3,$4,$5)`. The interpretation of this result is that if we know for a particular instance that the value of attributes `$1` and `$2` are `$3` and `$4` respectively, then we can guess the class of the instance to be `$5` with 80% accuracy. Of course, this accuracy is based only on the data we have seen but if this data is representative of all possible instances our estimate will hold for any new data.

2.5.2 Clustering

Consider the following simple clustering problem. We are given a set of two dimensional points, and we want to divide them into four regions, using one horizontal and one vertical line, such that each of the four regions contains at least 1/5 of all points. The points are given as the following relation:

- `points(x,y)`

Query Flock 2.8 expresses the problem of choosing a horizontal line at $Y = \$2$ and vertical line at $X = \$1$.

Query Flock 2.8 (Cluster Centers)

```
QUERY:
    answer1(P,Q) :- points(P,Q) AND P<=$1 AND Q<$2
    answer2(P,Q) :- points(P,Q) AND P>$1 AND Q<=$2
    answer3(P,Q) :- points(P,Q) AND P>=$1 AND Q>$2
    answer4(P,Q) :- points(P,Q) AND P<$1 AND Q>=$2
FILTER:
    COUNT(answer1(*)) >= COUNT(points(*))/5
    COUNT(answer2(*)) >= COUNT(points(*))/5
    COUNT(answer3(*)) >= COUNT(points(*))/5
    COUNT(answer4(*)) >= COUNT(points(*))/5
```

2.5.3 Sequence Analysis

One of the basic problems in sequence analysis is to identify a subsequence that occurs frequently in a given sequence of events.We model this problem with the following example.

Example 2.9 *Suppose we have a relation* `events` *that has information about some events and the sequence in which they occur:*

- `events(SequenceNumber, EventType)`

The problem is to find a frequent subsequence of event types `$1, $2, $3` *such that* `$2` *occurs within two events after* `$1`*, and* `$3` *occurs within two events after* `$2`*. Query Flock 2.9 expresses this problem, again taking frequent to mean at least 20 occurrences.*

Query Flock 2.9 (Frequent Subsequence)

QUERY:

```
answer(L) :-
     events(L, $1) AND
     events(M, $2) AND
     events(N, $3) AND
     L >= M-2 AND M >= N-2
     L < M AND M < N
```

FILTER:

```
COUNT(answer.L) >= 20
```

2.6 Concluding Remarks

In this chapter, we introduced the *query flock* framework for mining relational data. Query flocks allow a declarative formulation of large class of data mining queries. We presented example ranging form generalization of association rules, to clustering and classification. The next chapter details the other two features of query flocks, namely systematic optimization and integration with relational DBMS.

Chapter 3

Query Flock Plans

3.1 Introduction

As we showed in the last chapter, the query flock framework allows the declarative formulation of a large class of data mining problems. The present chapter is devoted to the other two main features of query flocks — systematic optimization and processing of data-mining queries and the integration of query flocks with relational DBMS.

The methods of optimization and processing are expressed as *Query Flock Plans (QFP)*. The key idea of QFP is to transform a complex query flock into an equivalent sequence of simpler queries such that each individual query can be processed efficiently. Thus, when query flocks are integrated with relational DBMS, the simpler queries are translated into SQL and executed at the DBMS.

In this chapter, we show how to find "good" query flock plans for a given query flock. The search for "good" QFP is akin to the search for logical query plans in query optimization[GMUW00] and consequently, in this thesis, we adapt and employ standard query optimization techniques such rule-based algebraic transformations[Gra87].

It is important to note that the bulk of the optimization techniques presented in this chapter apply directly to query flocks with certain types of filter conditions, called *monotone* conditions. The problem of optimizing and processing an arbitrary query flock with an arbitrary filter condition remains open. Indeed, we expect that, similar to standard query optimization, the problem of finding an optimal query flock plan for a query flock is intractable in the general case.

3.1.1 Chapter Organization

The rest of this chapter is organized as follows. In Section 3.2 we formally define query flock plans. Section 3.3 presents several optimization algorithms for generating "good" query flock plans. In Section 3.4 we describe the tightly-coupled integration of the query flock framework with relational DBMS and the implementation of query flock plans in SQL. Section 3.5 presents some experimental results. Section 3.6 concludes the chapter.

3.2 Definition of Query Flock Plans

In this section we present our definition of query flock plans. We begin by defining *auxiliary relations*, which are the most important ingredients of QFP. Then, we define the three types of steps that are the building blocks of query flock plans.

3.2.1 Auxiliary Relations

An auxiliary relation is a relation over a subset of the parameters of a query flock and contains candidate values for the given subset of parameters. The main property of auxiliary relations is that all parameter values that satisfy the filter condition are contained in the auxiliary relations. In other words, any value that is not in an auxiliary relation is guaranteed not satisfy the filter condition. Throughout the examples in this chapter we only consider filter conditions of the form $COUNT(ans) >= X$. However, our results and algorithms are valid for a larger class of filter conditions called *monotone* in [TUC$^+$98] or *anti-monotone* in [NLHP98]. For this class of filter conditions, according to Rule 2.2, the auxiliary relations can be defined with a subset of the goals in the query part of the query flock. For a concrete example, consider Query Flock 2.3 from Example 2.3:

Example 3.1 *An auxiliary relation ok_m for parameter $m can be defined as the result of the following query flock:*

Query Flock 3.1 (Common Medicines)

QUERY:

```
answer(P) :- treatments(P, $m)
```

FILTER:

```
COUNT(ans) >= 20
```

The result of this flock consists of all medicines such that for each medicine there are at least 20 patients treated with it. In order to illustrate the use of the auxiliary relation ok_m consider the result of Query Flock 2.3. The result consists of pairs of a medicine and a symptom such that there are at least 20 patients treated with the medicine, exhibiting the symptom, and not having symptom caused by their disease. Then, any medicine that appears in the result must appear in the ok_m. Thus, the auxiliary relation is a superset of the result of the original query flock projected on the parameters of the auxiliary relation..

3.2.2 Query Flock Plans

Intuitively, a query flock plan represents the transformation of a complex query flock into a sequence of simpler steps. This sequence of simpler steps represent the way a query flock is executed at the underlying RDBMS. In principle, we can translate any query flock directly in SQL and then execute it at the RDBMS. However, due to limitations of the current query optimizers, such an implementation will be very slow and inefficient. Thus, using query flock plans we can effectively pre-optimize complex mining queries and then feed the sequence of smaller, simpler queries to the query optimizer at the DBMS.

A query flock plan is a (partially ordered) sequence of operations of the following 3 types:

Type 1 Materialization of an auxiliary relation

Type 2 Reduction of a base relation

Type 3 Computation of the final result

The last operation of any query flock plan is always of type 3 and is also the only one of type 3.

Materialization of an auxiliary relation: This type of operation is actually a query flock that is meant to be executed directly at the RDBMS. The query part of this query flock is formed by choosing a safe subquery [Ull88] of the original query. The filter condition is the same as in the original query flock. Of course, there are many different ways to choose a safe subquery for a given subset of the parameters. We investigate several ways to choose safe subqueries according to some rule-based heuristics later in the chapter. This type of operation is translated into an SQL query with aggregation and filter condition.

For an example of a step of type 1, recall the query flock that materializes an auxiliary relation for \$m from Example 3.1. This materialization step can be translated directly into SQL as follows:

```
ok_m(Medicine) AS
SELECT Medicine
FROM treatments
GROUP BY Medicine
HAVING COUNT(Patient) >= 20
```

Reduction of a base relation: This type of operation is a semijoin of a base relation with one or more previously materialized auxiliary relations. The result replaces the original base relation. In general, when a base relation is reduced we have a choice between several reducers. Later in this chapter, we describes how to choose "good" reducers.

For an example of a step of type 2, consider the materialized auxiliary relation `ok_m`. Using `ok_m` we can reduce the base relation `treatments` as follows:

```
treatments_1(P) :- treatments(P,$m) AND ok_m($m)
```

This query can be translated directly into SQL as follows:

```
treatments_1(Patient,Medicine) AS
SELECT b.Patient, b.Medicine
FROM treatments b, ok_m r
WHERE b.Medicine = r.Medicine
```

Computation of the final result: The last step of every query flock plan is a computation of the final result. This step is essentially a query flock with a query part formed by the reduced base relations from the original query flock. The filter is the same as in the original query flock.

For an example of a step of type 3, consider Query Flock 2.3 with the `treatments` predicate replaced by the reduced base relation `treatments_1`:

Query Flock 3.2 (Side Effects with reduced diagnoses)

```
QUERY:
    answer(P) :-
        exhibits(P,$s) AND
        treatments_1(P,$m) AND
        diagnoses(P,D) AND
        NOT causes(D,$s)
FILTER:
    COUNT(answer.P) >= 20
```

This flock can be translated directly into SQL as follows:

```
result(Medicine,Symptom) AS
SELECT t.Medicine,e.Symptom
FROM exhibits e, treatments_1 t, diagnoses d
WHERE e.Patient = t.Patient
AND e.Patient = d.Patient
AND e.Symptom NOT IN
      (SELECT c.Symptom
       FROM causes c
       WHERE c.Disease = d.Disease)
GROUP BY t.Medicine,e.Symptom
HAVING COUNT(Patient) >= 20
```

3.3 Optimization Algorithms

In this section we present algorithms that generate *efficient* query flock plans. Recall that in our tightly-coupled mining architecture these plans are meant to be translated in SQL and then executed directly at the underlying RDBMS. Thus, we call a query flock plan *efficient* if its execution at the RDBMS is efficient. There are two main approaches to evaluate the efficiency of a given query plan: cost-based and rule-based. A cost-based approach involves developing an appropriate cost model and methods for gathering and using statistics. In contrast, a rule-based approach relies on heuristics based on general principles, such as applying filter conditions as early as possible. In this chapter, we focus on the rule-based approach to generating efficient query flock plans. The development of a cost-based approach is a topic of future research.

The presentation of our rule-based algorithms is organized as follows. First, we describe a general nondeterministic algorithm that can generate all possible query flock plans under the framework described in Section 3.2.2. The balance of this section is devoted to the development of appropriate heuristics, and the intuition behind them, that make the nondeterministic parts of the general algorithm deterministic. At the end of this section we discuss the limitations of conventional query optimizers and show how the query flock plans generated by our algorithm overcome these limitations.

3.3.1 General Nondeterministic Algorithm

The general nondeterministic algorithm can produce any query flock plan in our framework. Recall that a valid plan consists of a sequence of steps of types 1 and 2 followed by a final step of type 3. One can also think of the plan as being a sequence of two alternating phases: materialization of auxiliary relations and reduction of base relations. In the materialization phase we choose what auxiliary relations to materialize one by one. Then we move to the reduction phase or, if no new auxiliary relations have been materialized, to the computation of the final result. In the reduction phase we choose the base relations to reduce one by one and then go back to the materialization phase.

Before we described the nondeterministic algorithm in details we introduce the following two helper functions.

MaterializeAuxRel(Params, Definition) takes a subset of the parameters of the original query flock and a subset of the base relations. This subset forms the body of the safe subquery defining an auxiliary relation for the given parameters. The function assigns a unique name to the materialized auxiliary relation and produces a step of type 1.

ReduceBaseRel(BaseRel, Reducer) takes a base relation and a set of auxiliary relations. This set forms the reducer for the given base relation. The function assigns a unique name to the reduced base relation and produces a step of type 2.

We also assume the existence of functions *add* and *replace*, with their usual meanings, for sets and the function *append* for ordered sets. The nondeterministic algorithm is shown in Fig. 3.1

Algorithm 3.1
Input: Query flock QF
$Parameters$ – set of parameters of QF
$Predicates$ – set of predicates in the body of the query part of QF
Output: Query flock plan $QFPlan$

```
// Initialization
   BaseRels = Predicates
   AuxRels = ∅
   QFPlan = ∅
// Iterative Generation of Query Flock Plan
   while(true) do
(Q1)  choose NextStepType from {MATERIALIZE, REDUCE, FINAL}
      case NextStepType:
      MATERIALIZE: // Materialization of Auxiliary Relation
(Q2)     choose subset S of Parameters
(Q3)     choose subset D of BaseRels
         Step = MaterializeAuxRel(S, D)
         QFPlan.append(Step)
         AuxRels.add(Step.ResultRel)
      REDUCE: // Reduction of Base Relation
(Q4)     choose element B from BaseRels
(Q5)     choose subset R of AuxRels
         Step = ReduceBaseRel(B, R)
         QFPlan.append(Step)
         BaseRels.replace(B, Step.ResultRel)
      FINAL: // Computation of Final Result
         Step = MaterializeAuxRel(Parameters, BaseRels)
         QFPlan.append(Step)
         return QFPlan
      end case
   end while
```

Figure 3.1: General nondeterministic algorithm.

The number of query flock plans that this nondeterministic algorithm can generate is rather large. In fact, with no additional restrictions, the number of syntactically different query flock plans that can be produced by Algorithm 3.1 is infinite. Even if we restrict the algorithm to materializing only one auxiliary relation for a given subset of parameters, the number of query flock plans is more than double exponential in the size of the original query. Thus, we have to choose a subspace that will be tractable and also contains query flock plans that work well empirically. To do so effectively we need to answer several questions about the space of potential query flock plans. We have denoted these questions in Algorithm 3.1 with `(Q1) - (Q5)`.

(Q1) How to sequence the steps of type 1 and 2?

(Q2) What auxiliary relations to materialize?

(Q3) What definition to choose for a given auxiliary relation?

(Q4) What base relations to reduce?

(Q5) What reducer to choose for a given base relation?

There are two main approaches to answering `(Q1) - (Q5)`. The first one involves using a cost model similar to the one used by the query optimizer within the RDBMS. The second approach is to use rule-based optimizations. As we noted earlier, in this chapter we focus on the second approach.

In order to illustrate Algorithm 3.1, consider Query Flock 2.3.

Example 3.2 *In this example, we describe one possible query flock plan that can be generated by Algorithm 3.1 for the above query flock. The inputs are:*

- $Parameters = \{\$M, \$S\}$
- $Predicates = \{exhibits(P, \$S), treatments(P, \$M), diagnoses(P, D),$ $NOT\ causes(D, \$S)\}$

The initialization phase assigns the following values:

- $BaseRels = \{exhibits(P, \$S), treatment(P, \$M), diagnoses(P, D),$ $NOT\ causes(D, \$S)\}$

- $AuxRels = \emptyset$
- $QFPlan = \emptyset$

Then, we choose (Q1) *the first step of the plan to be* MATERIALIZE*. Next we choose* (Q2) *a subset of parameters S for which to materialize an auxiliary relation.*

- $S = \{\$S\}$

We also choose a subset of the base relations D that forms a safe subquery that defines an auxiliary relation for $\$S$.

- $D = \{exhibits(P, \$S)\}$

Then, we add a step to the query flock plan that materializes the auxiliary relation for the chosen parameter with the chosen definition. We also add the auxiliary relation to the set of auxiliary relations. Thus, at the end of the first iteration we have:

- $BaseRels = \{exhibits(P, \$S), treatment(P, \$M), diagnoses(P, D),$
  $NOT\ causes(D, \$S)\}$
- $AuxRels = \{ok_s(\$S)\}$
- $QFPlan = \{MaterializeAuxRel(\{\$S\}, \{exhibits(P, \$S)\}\}$

Then, we go back to (Q1) *and this time we choose the type of the next step to be $REDUCE$. In* (Q4)*, we choose a base relation B to reduce.*

- $B = \{exhibits(P, \$S)\}$

We also choose, in (Q5)*, a reducer R for $exhibits(P, \$S)$ from the auxiliary relations.*

- $R = \{ok\_s(\$S)\}$

Thus, at the end of the second iteration we have:

- $BaseRels = \{e\_1(P, \$S), treatment(P, \$M), diagnoses(P, D), NOT\ causes(D, \$S)\}$
- $AuxRels = \{ok\_s(\$S)\}$

Table 3.1: Example of a query flock plan produced by Algorithm 3.1.

Step	*Type*	*Result*	*QUERY*	*FILTER*
(1)	1	`ok_s($S)`	`ans_1(P) :- exhibits(P,$S)`	`COUNT(ans_1) >= 20`
(2)	2	`c_1(D,$S)`	`c_1(D,$S) :- causes(D,$S) AND ok_s($S)`	-
(3)	2	`e_1(P,$S)`	`e_1(P,$S) :- exhibits(P,$S) AND ok_s($S)`	-
(4)	3	`res($M,$S)`	`ans(P) :- e_1(P,$S) AND treatment(P,$M) AND diagnoses(P,D) AND NOT c_1(D,$S)`	`COUNT(ans) >= 20`

- $QFPlan = \{MaterializeAuxRel(\{\$S\}, \{exhibits(P, \$S)\}),$
  $ReduceBaseRel(\{exhibits(P, \$S)\}, \{ok\_s(\$S)\})\}$

For the third iteration we choose $REDUCE$ and reduce the base relation $causes(D, \$S)$. For the fourth and final iteration, we choose $FINAL$ and compute the final result. The state of the global variables for Algorithm 3.1 is shown below:

- $BaseRels = \{e\_1(P, \$S), treatment(P, \$M), diagnoses(P, D), NOT\ c\_1(D, \$S)\}$

- $AuxRels = \{ok\_s(\$S)\}$

- $QFPlan = \{MaterializeAuxRel(\{\$S\}, \{exhibits(P, \$S)\}),$
  $ReduceBaseRel(\{exhibits(P, \$S)\}, \{ok\_s(\$S)\}),$
  $ReduceBaseRel(\{causes(P, \$S)\}, \{ok\_s(\$S)\}),$
  $MaterializeAuxRel(\{\$S, \$M\},$
  $\{e_1(P, \$S), treatment(P, \$M), diagnoses(P, D), NOT\ c_1(D, \$S)\})\}$

Table 3.1 shows the example query flock plan in a concise table format.

3.3.2 Levelwise Heuristic

First, we address the question how to sequence the steps of types 1 and 2 ((`Q1`)) along with the questions what auxiliary relations to materialize ((`Q2`)) and what base relations to reduce ((`Q4`)). The levelwise heuristic that we propose is loosely fashioned after the highly successful a-priori trick [AIS93]. The idea is to materialize the auxiliary relations for all parameter subsets of size up to and including k in a levelwise manner reducing base relations after each level is materialized. So, starting at level 1, we materializing an auxiliary relations for every parameter. Then we reduce the base relations with the materialized auxiliary relations. At level 2, we materialize the auxiliary relations for all pairs of parameters, and so on. The general levelwise algorithm is formally described in Fig. 3.2.

The levelwise heuristic has also some important implications on the choice of definitions of auxiliary relations and the choice of reducer for base relations discussed in the next two section.

3.3.3 Choosing Definitions of Auxiliary Relations

When choosing definitions of auxliary relations ((Q3)) there are two main approaches single and group. In the single approach, we choose a definition for a single auxiliary relation without regard to any other choices. In the group approach, in contrast, we choose definitions for several auxiliary relations at the same time. Thus, we can exploit existing symmetries among the parameters or equivalences among syntactically different definitions. Regardless of the particular approach we only consider definitions that form *minimal* safe subquesies, not involving a cartesian product. The subquesies are minimal in a sense that eliminating any subgoal will either make the subquery unsafe or will turn it into a cartesian product.

The already chosen levelwise heuristic dictates the use of the group approach in our algorithm. We can take advantage of the fact that we are choosing definitions for all auxiliary relations for a given level *simultaneously*. Thus, it is rather straightforward to use symmetries among parameters and equivalences among subqueries to choose the smallest the number of definitions that cover all auxliary relations. We refer to this strategy as the *least-cover* heuristic.

3.3.4 Choosing Reducers of Base Relations

When choosing a reducer for a given base relation, we can employ two strategies. The first strategy is to semijoin it with the join of all auxliary relations that have parameters in common with the base relation. The second strategy is to semijoin it with all auxiliary relations that only have parameters appearing in the given base relation. With the second

Algorithm 3.2

Input: Query flock QF; K – max level
$Parameters$ – set of parameters of QF
$Predicates$ – set of predicates in the body of the query part of QF

Output: Query flock plan $QFPlan$

```
// Initialization
    BaseRels = Predicates
    QFPlan = ∅
// Levelwise Generation of Query Flock Plan
    for i = 1 to K do
        AuxRels_i = ∅
    // Materialization of Auxiliary Relations
        for all S ⊂ Parameters with | S |= i do
(Q3)            choose subset D of BaseRels
            Step = MaterializeAuxRel(S, D)
            QFPlan.append(Step)
            AuxRels_i.add(Step.ResultRel)
        end for
    // Reduction of Base Relations
        for all B ∈ BaseRels
(Q5)            choose subset R of AuxRels_i
            Step = ReduceBaseRel(B, R)
            QFPlan.append(Step)
            BaseRels.replace(B, Step.ResultRel)
        end for
    end for
// Computation of Final Result
    Step = MaterializeAuxRel(Parameters, BaseRels)
    QFPlan.append(Step)
    return QFPlan
```

Figure 3.2: General levelwise algorithm.

strategy we minimize the number of relations in the reduction joins while keeping the selectivity as high as possible. Again the use of the levelwise heuristic dictates our strategy choice. At the end of each level we have materialized auxiliary relations for all parameter subsets of the given size. Thus, the first strategy yields unnecessarily large reducers for every base relation at almost every level. Therefore, in our algorithm, we employ the second strategy.

3.3.5 K-Levelwise Deterministic Algorithm

Choosing the least-cover heuristic for `(Q3)` and the strategy outlined in Section 3.3.4 for `(Q5)` we finalize our algorithm that generates query flock plans. The formal description of the k-levelwise deterministic algorithm is shown in Fig.3.3.

The k-levelwise deterministic algorithm uses the following three helper functions.

GetMinDefs(Params,Preds) takes a set of parameters and a set a of predicates (query). The function returns a tuple where the first element is the set of parameters and the second element is the set of all minimal definitions (subqueries) for the auxiliary relation for the given set of parameters.

GetLeastCover(Set of (Params,Defs)) takes a set of tuples composed of a set of parameters and a set of definitions. The function returns the smallest set of definitions that covers all sets of parameters using equivalences among syntactically different definitions.

GetParams(Pred) takes a predicate and returns the set of parameters that appear in the given predicate.

The query flock plan produced by Algorithm 3.3 with $k = 1$ for the query flock from Example 2.3 is shown in Table 3.2.

3.3.6 Comparison with Conventional Query Optimizers

Recall that we use query flock plans to insure the efficient execution of query flocks at the underlying RDBMS. The shortcomings, with respect to query flocks, of conventional query optimizers are the fixed shape (left-deep trees) of their query plans and the fact that aggregation is usually done last. Query flock plans rectify these problems by using reduction

Algorithm 3.3

```
Input:  Query flock QF; K – max level
        Parameters – set of parameters of QF
        Predicates – set of predicates in the body of the query part of QF
Output: Query flock plan QFPlan
// Initialization
  BaseRels = Predicates; QFPlan = ∅
// Levelwise Generation of Query Flock Plan, up to level K
  for i = 1 to K do
    AuxRels_i = ∅; MinDefs_i = ∅
  // find all minimal definitions of auxiliary relations
    for all S ⊂ Parameters with | S |= i do
      MinDefs_i.add(GetMinDefs(S, BaseRels))
    end for
  // choose least cover of minimal definitions
    Cover_i = GetLeastCover(MinDefs_i)
  // for each definition in the cover add corresponding
  // auxiliary relations for all covered parameter subsets
    for all ⟨Def, CoveredParamSets⟩ ∈ Cover_i do
      for all S ∈ CoveredParamSets do
        Step = MaterializeAuxRel(S, Def)
        AuxRels_i.add(Step.ResultRel)
      end for
    // materialize the shared definition only once
      QFPlan.append(Step)
    end for
  // Reduction of Base Relations
    for all B ∈ BaseRels do
      R = ∅
    // choose reducer for base relation
      for all A ∈ AuxRels_i do
        if GetParams(A) ⊂ GetParams(B) then
          R.add(A)
      end for
      Step = ReduceBaseRel(B, R)
      QFPlan.append(Step)
      BaseRels.replace(B, Step.ResultRel)
    end for
  end for
// Computation of Final Result
  Step = MaterializeAuxRel(Parameters, BaseRels)
  QFPlan.append(Step)
  return QFPlan
```

Figure 3.3: K-Levelwise deterministic algorithm.

Table 3.2: Query flock plan produced by Algorithm 3.3 with $K = 1$.

Step	*Type*	*Result*	*QUERY*	*FILTER*
(1)	1	`ok_s($S)`	`ans_1(P) :- exhibits(P,$S)`	`COUNT(ans_1) >= 20`
(2)	1	`ok_m($M)`	`ans_2(P) :- treatment(P,$M)`	`COUNT(ans_2) >= 20`
(3)	2	`c_1(D,$S)`	`c_1(D,$S) :- causes(D,$S) AND ok_s($S)`	-
(4)	2	`e_1(P,$S)`	`e_1(P,$S) :- exhibits(P,$S) AND ok_s($S)`	-
(5)	2	`t_1(P,$M)`	`t_1(P,$M) :- treatment(P,$M) AND ok_m($M)`	-
(6)	3	`res($M,$S)`	`ans(P) :- e_1(P,$S) AND t_1(P,$M) AND diagnoses(P,D) AND NOT c_1(D,$S)`	`COUNT(ans) >= 20`

of base relations to compensate the fixed shape of the query plan and auxiliary relations to use aggregation on partial results as early as possible

The problem of including aggregation in query optimization is studied in [YL95, CS96, CS94]. In these papers, aggregation is pushed down, (or sometimes up), the query plan tree. The key difference with our work is that we use aggregation on a subset of the original query, and the result is used to reduce the size of intermediate steps. Eventually the aggregation must be performed again but we have gained efficiency by having much smaller intermediate results.

3.4 Integration with Relational DBMS

There are three different ways in which data-mining systems use relational DBMS. They may not use a database at all, be loosely coupled, or be tightly coupled. We have chosen the tightly-coupled approach that does (almost) all of the data processing at the database. Before we justify our choice, we discuss the major advantages and drawback of the the other two approaches.

Most current data mining systems do not use a relational DBMS. Instead they provide their own memory and storage management. This approach has its advantages and disadvantages. The main advantage is the ability to fine-tune the memory management algorithms with respect to the specific data mining task. Thus, the data mining systems can achieve optimal performance. The downside of this database-less approach is the lost opportunity to leverage the existing relational database technology developed in the last couple of decades. Indeed, conventional DBMS provide various extra features, apart from good memory management, that can greatly benefit the data mining process. For example, the recovery and logging mechanisms, provided by most DBMS, can make the results of long computations durable. Furthermore, concurrency control can allow many different users to utilize the same copy of the data and run data mining queries simultaneously.

Some data mining systems use a DBMS but only to store and retrieve the data. This loosely-coupled approach does not use the querying capability provided by the database which constitutes both its main advantage and disadvantage. Since the data processing is done by specialized algorithms their performance can be optimized. On the other hand, there is still the requirement for at least temporary storage of the data once it leaves the database. Therefore, this approach also does not use the full services offered by the DBMS.

The tightly-coupled approach, in contrast, takes full advantage of the database technology. The data are stored in the database and all query processing is done locally (at the database). The downside of this approach is the limitations of the current query optimizers. In was shown in [STA98] that performance suffers greatly if we leave the data mining queries entirely in the hands of the current query optimizers. Therefore, we need to perform some optimizations *before* we send the queries to the database, taking into account the capabilities of the current optimizers. To achieve this, we use the algorithms that generate query flock plans as an *external optimizer* that sits on top of the DBMS. The external optimizer effectively breaks a complex data mining query into a sequence of smaller queries that can be executed efficiently at the database. This architecture is shown in Fig. 3.4.

The external optimizer can be a part of larger system for formulating data mining queries such as query flocks. The communication between this system and the database can be carried out in ODBC or JDBC.

3.5 Experimental Results

Our experiments are based on real-life health-care data. Below we describe a representative problem and the performance results.

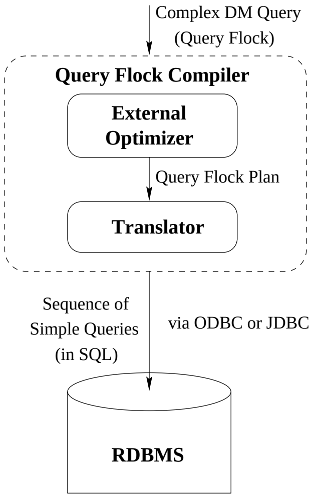

Figure 3.4: Tightly-coupled integration of data mining and DBMS.

Consider a relation `Diagnoses(PatientID,StayCode,Diagnose)` that contains the diagnoses information for patients during their stays at some hospital. Another relation, `Observe(PatientID,StayCode)`, contains the pairs of `PatientID` and `StayCode` for patients that are kept for observations for less than 24 hours. The rest of the patients are admitted to the hospital. Consider the following problem.

Find all pairs of diagnoses such that:

1. There are at least N patients diagnosed with the pair of diagnoses
2. At least one of them is an observation patient

We can express this problem naturally as a query flock:

QUERY:

```
ans(P,S) :- Diagnoses(P,S,$D1) AND
            Diagnoses(P,S,$D2) AND
            Diagnoses(Q,T,$D1) AND
            Diagnoses(Q,T,$D2) AND
            Observe(Q,T) AND
            $D1 < $D2
```

FILTER:

```
COUNT(ans) >= N
```

This problem is important to the hospital management because the reimbursement procedures and amounts for admitted and observation patients are different. Thus, management would like to identify some exceptions to the general trends, find their causes, and investigate them further for possible malpractice or fraud.

The `Diagnoses` relation contains more than 100,000 tuples, while the `Observe` relation contains about 8,000 tuples. We compared the performance of the 1-levelwise and 2-levelwise algorithms as well as the direct approach where the query flock is directly translated into SQL. We used a standard installation of ORACLE 8.0 running under Windows NT. The results are shown in Fig. 3.5.

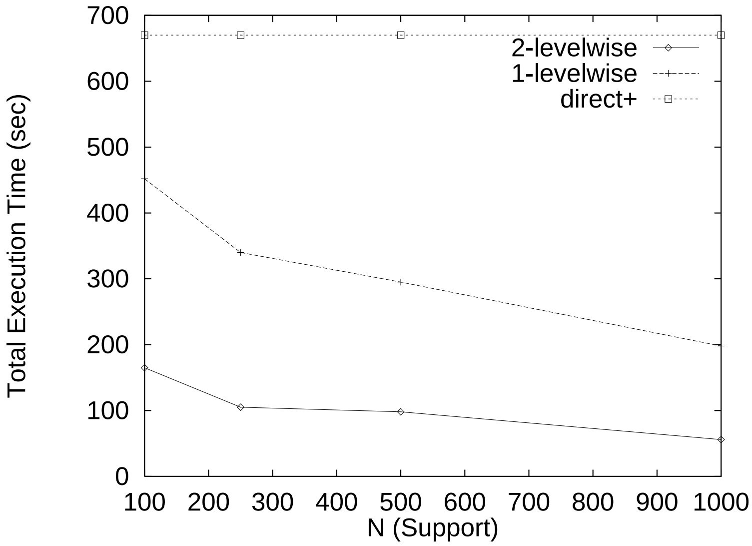

Figure 3.5: Performance results on health-care data.

For this dataset, the 2-levelwise algorithm outperforms the 1-levelwise algorithm more than 3 times. This result is somewhat surprising because the two parameters \$D1 and \$D2 are symmetric (excluding the inequality) and thus, only one relation is materialized at level 1. However, the reduced base relation `Diagnoses` after level 1 is still rather large and the computation of the final result at this stage is much slower than materializing the auxiliary relation for the pair of parameters.

As expected, both algorithms perform much better than the direct approach where we translate the query flock directly in SQL. Infact, the actual translation did not finish executing in a reasonable amount of time. Thus, we had to augment the direct translation, hence *direct+*, with a preliminary step where we joined the `Observe` and `Diagnoses` relations. This step had the effect of reducing the size of the relations for two of the four `Diagnoses` predicates and eliminating the `Observe` predicate.

3.6 Concluding Remarks

In this chapter, we introduced query flock plans that transform a complex query flock into an equivalent sequence of simpler queries that can be executed efficiently as a unit. We also showed that query flock plans can be integrated with relational DBMS in a tightly coupled manner.

Chapter 4

Mining Semistructured Data

4.1 Introduction

The advent of the Internet and its rapid proliferation are making an ever increasing number of information sources available electronically to the casual user. These sources export data in a variety of formats with widely varying degrees of structure, regularity, and consistency. While the data can rarely fit the rigid requirements of the relational model, it often has some underlying, implicit structure. We call such data *semistructured.*

The World Wide Web (WWW) provides a perfect example of *semistructured* data. Consider the home pages of students at some university. Most of these pages will have the names and addresses of their respective owners. Some pages will have links to the home pages of other students, university clubs and organizations, home towns, favorite sports teams, etc. Certainly, these pages will not conform to some predefined fixed schema in the relational sense. Furthermore, they are interconnected via hyperlinks. And yet, there is some inherent structure that underlines these pages. The common structure may come about by way of students sharing page templates, being in the same environment, having common interests, etc.

Of course, we cannot expect to find only one kind of student home pages. There are bound to be many different kinds corresponding to the different types of students. Furthermore, most students will belong to several different types. For example, consider a European engineering graduate student (such as the author). The home page of this student might have some soccer links (as European), links to the pages of his advisor and research group (as an engineering student), as well as a link to his undergraduate college (as a graduate student).

Another source of semistructured data is information integration [CGMH+94]. Consider the increasingly popular idea of comparison shopping across many different vendors on the Internet. Even if we assume that each individual shopping site has a regular structure, the data, resulting from the integration of several thousands of them, will certainly be somewhat irregular. On the one hand, different shopping sites present different kinds of information about the same product. On the other hand, each product is sold on a different set of shopping sites. Thus, in the integrated data each product may have different kinds of information, unlike that of any other product. However, we can also expect that similar products will have similar, but not identical, kinds of information. Identifying such similar products and their common information is of great value to the integrators. Using the mined common information, the integrators, can present a common set of features for each of the different product categories. Thus, customers can search and compare similar products effectively across many different vendors.

These two examples illustrate the importance of discovering the implicit structure of semistructured data. In fact, structure discovery is the most important application of data mining for semistructured data. Because of the nature of semistructured data, the discovered structure may be imprecise or approximate. There is a tradeoff between the precision of the discovered structure and its conciseness. This tradeoff depends on the particular application as well as the actual semistructured data. For example, if we group together all products sold on the web, they will have very few common features (such as name and price). On the other hand, computers will have a lot more common features such as processor and memory.

Recently, the emerging *eXtensible Markup Language* (XML) has come into prominence for describing and exchanging certain types of data. Most of the XML data can arguably be considered semistructured even though it has a corresponding *Data Type Definition* (DTD). The main reason is that a DTD encodes *all* possible instances, often infinitely many, of a particular type of data whereas any given data instance has only some of the characteristics defined by the DTD. Thus, it is not uncommon for a DTD to be much larger in size than the actual data instances. In a sense, a DTD serves as grammar, whereas the data instances are the actual words and sentences. Thus, it makes sense to identify different kinds of similar sentences even though they all share the same grammar.

4.1.1 Chapter Organization

The rest of this chapter is organized as follows. Section 4.2 elaborates our definition of semistructured data and presents several data models. Section 4.3 describes the main goal of data mining semistructured data which is the discovery of its implicit structure. This section also gives an overview of our research contributions which are detailed in Chapters 5 and 6. Related work is discussed in Section 4.4. Section 4.5 concludes this chapter.

4.2 Semistructured Data Defined

There is a plethora of data available on the web, ranging in structure from rigid relational form to just plain text. A great portion of the data, however, falls somewhere in between. Recently, the term *semistructured data* has emerged to described such data. This is not a precise definition and the distinction between structured, semistructured, and unstructured data are often blurred and somewhat arbitrary. The consensus of the literature [ABS99, Abi97, Bun97], is that *semistructured data* has one or more of the following characteristics:

- no fixed schema known in advance;
- implicit structure that is irregular, incomplete, or partial;
- heterogeneous;
- hierarchical (nested);

Figure 4.1 shows a simple example of semistructured data about three students at United University. The data model will be explained in details in the next section; for now, it suffices to say that circles with text inside correspond to objects and their unique identifiers, arrows with text connected to labeled links between objects, and text below circles corresponds to object values.

The data, shown in Figure 4.1, exhibits all of the characteristics of semistructured data. Certainly, it does not conform to any fixed schema since each of the three Student objects has a different set of incoming and outgoing links. However, there is some implicit structure because each Student object has links to Name and Address objects as well as a link to another Student object. On the other hand, this structure is only partial because two Student objects have a Friend links, while the third one has a Roommate link. The data

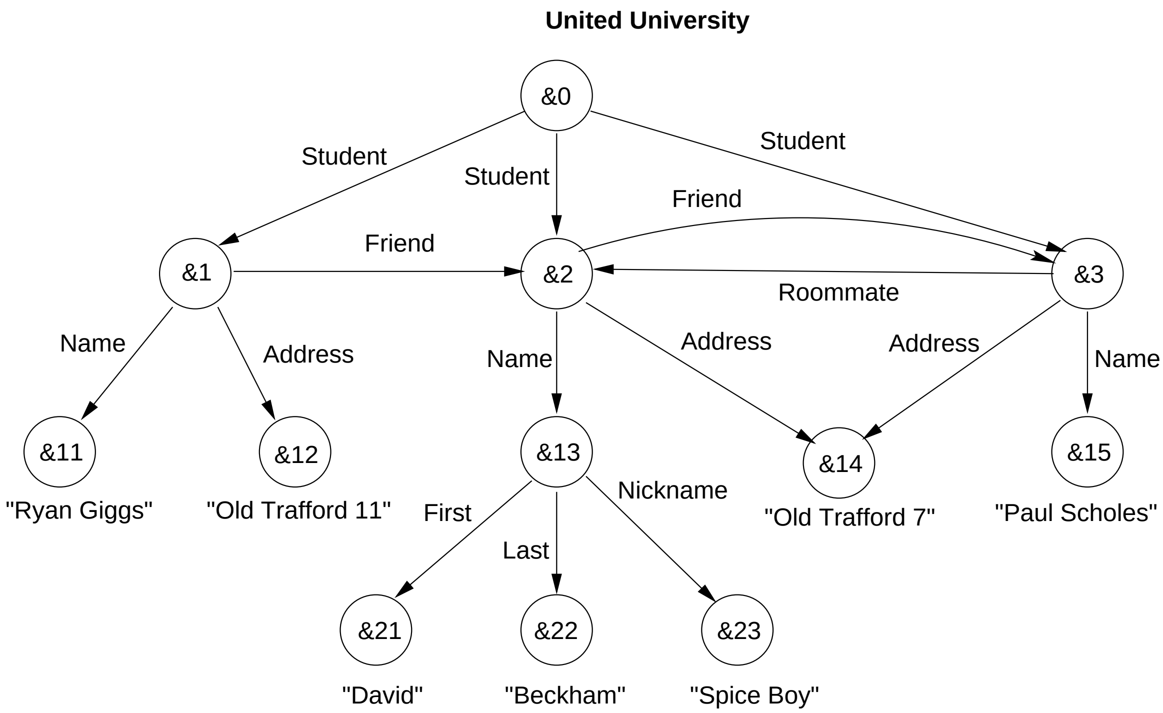

Figure 4.1: Simple example of semistructured data

is heterogeneous because two of the three Name objects have only text values whereas the third one has links to other objects. Finally, the data is nested because of the links among objects.

4.2.1 Data Models

The nature of semistructured data dictates that its data models should be *lightweight* and *self-describing*. We use the term *lightweight* to indicate that there is no explicit schema or typing information in the data model. The lack of schema and types makes it possible to handle data from newly discovered sources, which is often the case on the Web [Abi97], where new data source appear everyday and existing ones change their format. The semistructured data models need to incorporate implicitly the partial, incomplete, or irregular structure of the data and thus are *self-describing*.

Most semistructured data models follow a minimalistic approach that is consistent with being lightweight and self-describing. The majority of the literature follows one of the two independent, but similar, proposals that are based on labeled directed graphs [AQM+96, BDHS96].

The Object Exchange Model (OEM) The model consists of a directed graph with labeled edges. An earlier version of OEM put the labels on the vertices [PGMW95] which resulted in decreased flexibility of the model. In the "new" OEM [AQM+96] vertices correspond to objects and edges along with their respective labels correspond to object references. Each object has a unique object identifier. There two kinds of objects in OEM: *atomic* and *complex*. Atomic objects take on a value from one of the basic atomic types such as string, integer, html, etc. The value of a complex object is the set of its *object references*. An example of semistructured data represented in OEM is shown in Figure 4.1. Consider the object with object identifier &1. It is a complex object and its value is the set of three object references: $\{\langle Name, \&11\rangle, \langle Address, \&12\rangle, \langle Friend, \&2\rangle\}$. The object with object identifier &11 is an atomic object and its value is the string *Ryan Giggs*.

The Data Model of UnQL The model is based on rooted, labeled graphs. There are two main differences between this data model [BDHS96] and OEM. The UnQL data model puts all labels on the edges, i.e. vertices have no values associated with them. Thus, an atomic value is represented by a label on an edge to a leaf node in this model. The second difference is that there are no object identities in the UnQL data model. Tree markers and bisimulation are used instead. Regardless of these two difference the UnQL data model is very similar to OEM.

The eXtensible Markup Language (XML) Technically, XML is not a data model. Rather, it is a textual language, based on nested tagged elements [XML98], for representing and exchanging data on the Web. Nevertheless, semistructured data is the primary target of XML. There are several substantial differences between XML and the two data model presented above. Firstly, XML imposes an ordering on the object references that is absent from the graph-based models. Secondly, there is a distinction in XML between *attributes, subobjects,* and *object references*. In the graph-based models all of them will be considered as object references. Thirdly, an XML data can be accompanied by a DTD that serves as a schema for the data. A detailed discussion of these differences appears in [GMW99]. Figure 4.2 shows one possible XML representation of the semistructured data shown in Figure 4.1.

```
<?xml version="1.0" standalone="yes"?>
<University name="United University" id="&0">
   <Student id="&1">
     <Name id="&11">Ryan Giggs</Name>
     <Address id="&12">Old Trafford 11</Address>
     <Friend idref="&2"/>
   </Student>
   <Student id="&2">
      <Name id="&13">
        <First id="&21">David</First>
        <Last id="&22">Beckham</Last>
        <Nickname id="&23">Spice Boy</Nickname>
      </Name>
      <Address id="&14">Old Trafford 7</Address>
      <Friend idref="&3"/>
    </Student>
    <Student id="&3">
      <Name id="&15">Paul Scholes</Name>
      <Roommate idref="&2"/>
      <Address idref="&14"/>
    </Student>
</University>
```

Figure 4.2: A simple example of semistructured data in XML.

4.3 Mining for Structure

The lack of explicit schema in semistructured data makes it easy to generate, collect, and maintain but hard to store efficiently, browse, and query. Consider the example data about United University from Figure 4.1. Anyone who uses this data will first need to know what kinds of information are provided about students. In particular, users need to know that the data contains the names, addresses, and friends of students but not their emails or telephone numbers. In a relational or object-oriented database this meta-information can be discerned from the schema of the data. In semistructured data, however, the only way to discover all the different kinds of information in the database, is to examine each and every piece of data.

Furthermore, suppose that a user is interested in finding information about a particular student, say "David Beckham." In order to write this query, the user needs to know how the names of the students are being represented. In the absence of a schema, the user may start by examining a few student objects. If the user happens to come across objects with identifiers &1 and &3 first, and decides that the name is stored as a Name subobject, then the result of the user's query will be empty because the name "David Beckham" is stored differently.

The point is that in semistructured data objects do not have to conform to any schema, and there is no way of knowing their structure unless we examine all of them. Therefore, having some kind of a structural summary will be of a great value to browsing and querying semistructured data.

There are many benefits of having explicit structure for semistructured data:

- Data Browsing: Before users decide to query a particular semistructured database, they need to know whether the database contains any information relevant to their particular query. An explicit structure will serve as a summary or a table of contents of the database.

- Query Writing: When querying semistructured data, users need to express paths through the data that meet certain conditions. The use of wild-cards in such queries is inevitable, but having an explicit structure can help users minimize the wild-cards and, thus focus their search more effectively.

- Query Optimization: Executing queries over semistructured data without any other information may result in examining the whole database for every query. An explicit structure will allow more efficient execution as well as indexing.

- Efficient Storage: In order to cluster objects that tend to be accessed together the system needs to know something about the structure of the data. Having an explicit structure will allow the system to identify "similar" objects and store them together.

4.3.1 Our Research Contributions

We propose two different approaches to discovering the inherent structure of semistructured data. Both methods have their respective advantages and disadvantages for certain applications.

Our first approach is based on the concept of a *representative object* for a semistructured data source. The main property of a representative object (RO) is that it accurately describes all objects within the given data source. Thus, the RO is very similar to a DTD for an XML data source. However, unlike a DTD, which is given apriori, a RO is derived from the data source. Representative objects are implemented in the *Lore* DBMS [QRS+95] as *DataGuides* [GW97]. Chapter 5 is devoted entirely to the representative object concept and details several different methods of constructing a RO for a given semistructured data source.

The major disadvantage of the RO approach is the implicit assumption of perfect data. Since no errors are allowed, a single misspelling of a label may result in a doubling of the size of the RO. Our second approach, detailed in Chapter 6, rectifies this problem by discovering an *approximate* schema for a semistructured data source. In particular, we consider the tradeoff between precision and conciseness and conjecture that for many data sources with inherent structure there is a *natural* schema or a small range of natural schemas. Another distinguishing feature of our approach is that it allows objects to reside in incomparable types. We view this as an essential requirement because of the very nature of semistructured data.

4.4 Related Work

The rapid proliferation of the World Wide Web has spurred a flurry of research activities focused on semistructured data. The research topics range from novel query languages [AQM+97, BDHS96, FFLS97, BK94] to specialized database management systems designed for semistructured data [QRS+95, BCK+94, FFLS97]. There have been several different approaches to mining structure from semistructured data.

The problem of finding frequent common substructures for a collection of semistructured objects is considered in [WL97] and [WL00]. The treatment of this problem is fashioned after finding large itemsets for market basket data [AIS93]. The common substructures (corresponding to itemsets in the market basket problem) are called *tree-expressions*. They are, in fact, labeled, directed trees augmented with a wild-card label that matches any label. The size of a tree-expression is measured by its number of leaf nodes. The mining algorithm in the paper employs level-wise pruning techniques to efficiently generate all frequent tree-expressions. Thus, smaller frequent tree-expressions are combined to form the larger frequent-candidate tree-expressions. This approach works well only when the objects in the semistructured database are layered, have relatively shallow depth, and contain no cycles. For many real datasets, however, these condition do not hold. Furthermore, the use of absolute support to determine common substructures has been shown in [NAM97] not to be appropriate for semistructured data.

[GM99] considers the problem of finding the common structure of HTML pages using a model based on tuples and sets. The paper introduces the concept of *mark-up encodings* which describes mappings from data trees to (HTML) strings. The proposed approach is appropriate only for fairly well-structured data. Another limitation of this approach is the requirement that the HTML pages conform to a single schema. Thus, having several different types of web pages will likely result in finding a trivial common structure. The assumption of well-structured data is also made in [Ade98, AD99] which uses user interaction and guidance to mine common substructures of semistructured objects.

Another approach [BDFS97] picks a logic language that can be used for algebraic query optimization and transformations but does not account for possible errors in the data.

The implementation of the *representative object* concept as *DataGuides* in the context of the Lore DBMS is discussed in [GW97]. The paper also shows how to incorporate sample values and other statistical information in the DataGuides and use this information in query optimization. The DataGuides are also used in Lore as a backend of a query-by-example GUI. The notion of *approximate* DataGuides is discussed in [GW99]. The main idea is to allow certain kinds of inaccuracies in order to decrease the size of the DataGuide. However, the paper does not consider the tradeoff between the amount of introduced inaccuracies and the corresponding saving in the size of the DataGuide.

4.5 Concluding Remarks

In this chapter, we introduced the concept of *semistructured data* which is hierarchical data that has no fixed schema, known apriori, but has implicit structure that is irregular or incomplete. We motivated the problem of mining structure or schema form semistructured data and proposed several practical uses of the discovered structure in storing, browsing, and querying semistructured data. We gave an overview of our research contribution to this topic which are detailed in the next two chapters. We also discussed related work.

Chapter 5

Representative Objects

5.1 Introduction

This chapter presents the concept of a *representative object*, which is a tool that facilitates querying and browsing of semistructured data. The lack of external schema information currently makes browsing and querying semistructured data sources inefficient at best, and impossible at worst. For instance, a user finding a "person" object in a traditional object-oriented system would know the structure of its subobjects or fields. As an example, the class declaration for the object might tell us that each person-object has two subobjects: first-name and last-name. In a semistructured world, some person-objects might have subobjects with first name and last name. Other person-objects might have a single subobject with a single name as value, or a single "name" subobject that itself has subobjects first- and last-name. Yet another person-object might have a middle-name subobject, while others have no name at all or have two name subobjects, one of which is a nickname or alias.

There are several ways to deal with the lack of fixed schema. If the semistructured data is somewhat regular but incomplete, then an object-oriented or relational schema can be used (along with null values) to represent the data. This approach fails, however, if the semistructured data is very irregular. Then, trying to fit the data into a traditional database form will either introduce too many nulls or discard most of the information [Ull88].

In this chapter we introduce the *representative object* concept. The representative object allows browsers to uncover the inherent schema(s) in semistructured data. Representative objects are implemented in the *Lore* DBMS as *DataGuides* [GW97]. Representative objects provide not only a concise description of the structure of the data but also a convenient way of querying it. The next subsection describes the primary uses of the representative objects.

5.1.1 Motivating applications

- Schema discovery: To formulate any meaningful query for a semistructured data source we need first to discover something about how the information is represented in the source. Only then can we pose queries that will match some of the source's structure. Representative objects give us the needed knowledge of the source's structure.

- Path queries: When querying semistructured data, we often need to express paths through the semistructured objects that meet certain conditions, e.g., the path ends in a "name" object, perhaps going through one or more other objects. Expressing such paths requires "wild cards" — symbols that stand for any sequence of objects or objects whose class names (which we call "labels") match a certain pattern. However, when queries have wild-card symbols in them, searching the entire structure for matches is infeasible. The representative object can significantly reduce the search.

- Query Optimization: We can optimize some queries or subqueries by noticing from the representative object that their results must be empty.

5.1.2 Chapter Organization

In Section 5.2, we introduce our data model and define several terms and functions regarding the semistructured nature of the data, including the OEM (Object-Exchange Model) used in the Tsimmis and Lore projects at Stanford. Then in Section 5.3, we define both full representative objects (FROs), which provide a description of the global structure of the data, and the degree-k representative objects (k-ROs), which provide a description of the local aspects of the data, considering only paths (in the object-subobject graph) of length k. Section 5.4 describes an implementation of FROs as objects in OEM and an algorithm for extracting the relevant information from them. We also consider minimal FROs, which allow us to answer schema queries most efficiently. In Section 5.5, we present a method based on determinization and minimization of nondeterministic finite automata for construction of a minimal FRO in OEM. Section 5.6 describes the construction and use of the simplest

k-RO, the case $k = 1$. Section 5.7 present an automaton-based approach to building a k-RO for $k > 1$. Section 5.8 concludes this chapter.

5.2 Preliminaries

In this section we describe the data model used in this chapter. The object-exchange model (OEM) [AQM+96] is designed specifically for representing semistructured data for which the representative objects are most applicable and useful. The OEM described in [AQM+96] that we use is a modification of the original OEM introduced in [PGMW95]. We then define several terms that are related to the structure of the objects in OEM. We also define two functions that form the basis of the representative object definitions.

5.2.1 The Object-Exchange Model

Our data model, OEM, is a simple, self-describing object model with nesting and identity. Every object in OEM consists of an *identifier* and a *value*. The *identifier* uniquely identifies the object. The *value* is either an atomic quantity, such as an integer or a string, or a set of object references, denoted as a set of $\langle label, id \rangle$ pairs. The *label* is a string that describes the meaning of the relationship between the object and its subobject with an identifier *id*. Objects that have atomic values are called *atomic objects* and objects that have set values are called *complex objects*. We can view OEM as a graph where the vertices are the objects and the labels are on the edges (object references).

Figure 5.1 shows a segment of information about the top soccer league (The Premiership) in England. Each circle along with the text inside it represents an object and its identifier. The arrows and their labels represent object references.

We will use the notations $identifier(o)$ and $value(o)$ to denote the identifier and value of the object o. We will also use the notation $object(id)$ (or $obj(id)$ for short) to denote the unique object with an identifier *id*.

5.2.2 Simple Path Expressions and Data Paths

Intuitively, a *simple path expression* is a sequence of labels. A *data path* is a sequence of alternating objects and labels that starts and ends with an object. This sequence has the property that for every two consecutive objects the value of the first object contains an

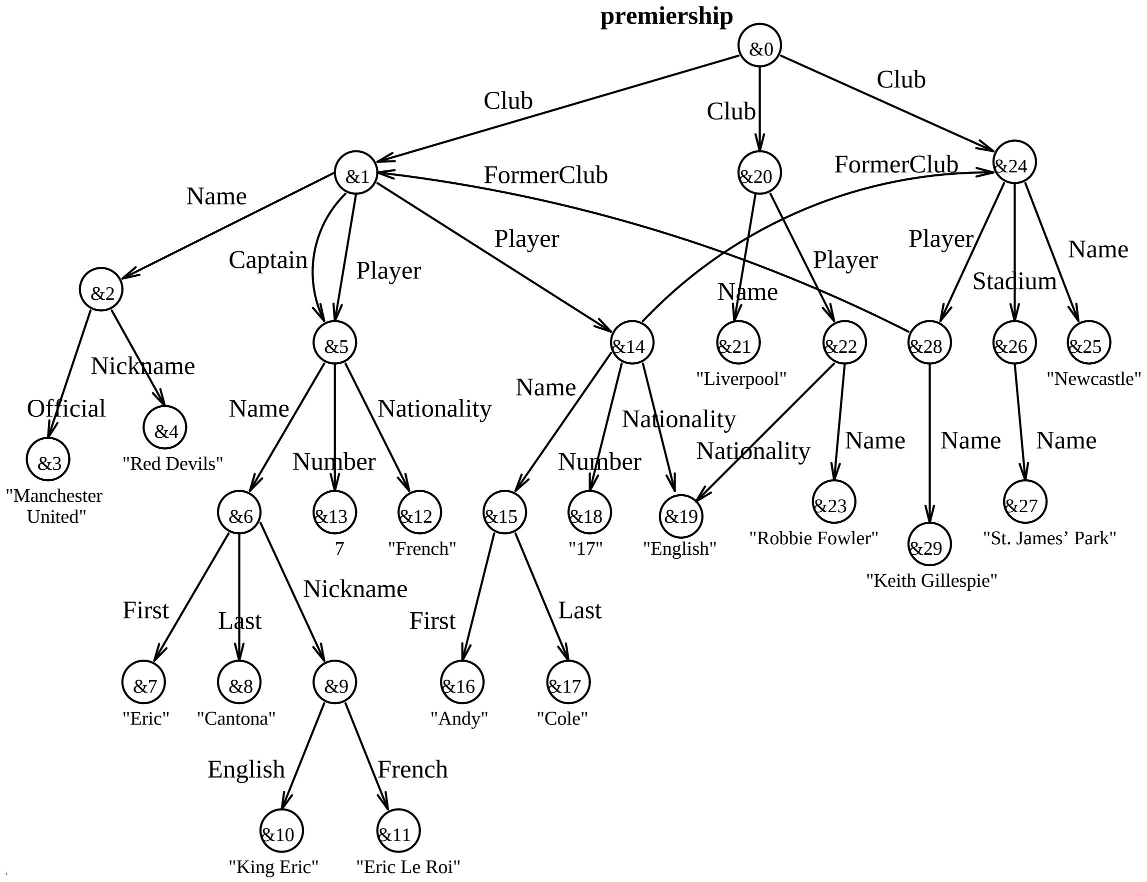

Figure 5.1: The premiership object.

object reference to the second object, labeled with the label that is between the two objects in the given sequence. Formally, we have the following two definitions:

Definition 5.1 *Let l_i be a label (of object references) for $i = 1 \cdots n, n \geq 1$. Then $pe = l_1.l_2 \cdots l_n$ is a* simple path expression *of length n. The special symbol ϵ denotes a path expression of length 0.*

Definition 5.2 *Let o_i be an object for $i = 0..n$ and l_i be a label for $i = 1 \cdots n$ such that we have $\langle l_i, identifier(o_i)\rangle \in value(o_{i-1})$ for $i = 1 \cdots n, n \geq 0$. Then $p = o_0, l_1, o_1, l_2 \cdots l_n, o_n$ is a* data path, *of length n.*

We introduce the following terminology regarding simple path expressions and data paths.

- A data path $p = o_0, l_1 \cdots l_n, o_n$ *originates from* or is *rooted at* the object o_0.
- An object o_1 is *within* an object o if $\exists$ a data path originating from o and ending with o_1.
- A data path p is *within* an object o if p originates from an object within o.
- A data path $p = o_0, l_1 \cdots l_n, o_n$ is an *instance of* the simple path expression $pe = l_1.l_2 \cdots l_n$.

Remark 5.1 *Note that we allow data paths of length 0. Any data path of length 0 consists of a single object and is an instance of ϵ, the path expression of length 0.*

Example 5.1 *To illustrate the above terms consider the premiership object from Figure 5.1.*

- *The simple path expression* Player.Number *has two* instance data paths *within the premiership object:*
  - *obj*(&1),Player,*obj*(&5),Number,*obj*(&13)
  - *obj*(&1),Player,*obj*(&14),Number,*obj*(&18)
- *The simple path expression* Name.First *has two* instance data paths *within the premiership object:*
  - *obj*(&5),Name,*obj*(&6),First,*obj*(&7)
  - *obj*(&14),Name,*obj*(&15),First,*obj*(&16)
- *Consider the following two data paths*
  - *obj*(&1),Player,*obj*(&14),FormerClub,*obj*(&24)
  - *obj*(&24),Player,*obj*(&28),FormerClub,*obj*(&1)

  From the existence of the first path we have that obj(&1) is within obj(&24). The existence of the second path means that obj(&24) is within obj(&1), Thus, there is a cycle within the premiership object.

  On the other hand, obj(&20) is not within obj(&1) because there is no path that originates at obj(&1) and ends in obj(&20).

5.2.3 Continuations

Continuation functions form the basis of the representative object definitions presented in the next section. However, they arise naturally when we consider schema discovery of semistructured data. We briefly describe the schema discovery process before we give the formal definitions of the continuation functions.

Consider a semistructured object o represented in OEM. Suppose that we are interested in the structure (schema) of the object, i.e., we want to perform schema discovery. By schema discovery we mean exploring o by moving (navigating) from an object to its subobjects and keeping track of the labels of the object references that we traverse. By following a given sequence of labels (a simple path expression) we can get, in general, to zero, one, or more objects within o. At this point we want to know the labels of the links we could immediately traverse if we continue our navigation. We also want to know if we might not be able to continue navigating, i.e., we have reached an atomic object, but we are not (yet) interested in the specific value of the atomic object. These observations motivate to the following definition.

Definition 5.3 *Let o be an object in OEM and $pe = l_1.l_2 \cdots l_n$ a simple path expression, $n \geq 0$. We define $continuation(o, pe)$ as follows.*

- *$continuation(o, pe) \supseteq \{l \mid \exists$ a data path $p = o, l_1, o_1 \cdots l_n, o_n, l, o_{n+1}$ that is an instance of $pe.l\}$.*
- *$continuation(o, pe) \supseteq \{\perp \mid \exists$ a data path $p = o, l_1, o_1 \cdots l_n, o_n$ that is an instance of pe and o_n is an atomic object$\}$.*
- *$continuation(o, pe)$ contains nothing else.*

If we view OEM as a graph, Definition 5.3 translates into the following. The continuation of o and the simple path expression ϵ is the set of the labels on all outgoing edges from o. The continuation of o and a simple path expression pe of length $n \geq 1$ is obtained as follows. First, we traverse all possible paths of length $n+1$ starting at o, such that at the ith step, $1 \leq i \leq n$, we pick an edge labeled with the ith label in pe. At the last, $n+1$st step we pick any edge. The continuation of o and pe is the set of all labels on the edges that we picked last plus $\perp$ if in any of the traversals we made the first n steps but could not make the $n+1$st step because we ended up at a vertex with no outgoing edges (corresponding to an atomic object).

Example 5.2 *Consider the premiership object from Figure 5.1. The following examples illustrate Definition 5.3.*

- $continuation(premiership, \epsilon) = \{Club\}$
- $continuation(premiership,\ Club) = \{Name,\ Player,\ Stadium,\ Captain\}$
- $continuation(premiership,\ Club.Name) = \{Official,\ Nickname,\ \perp\}$
- $continuation(premiership,\ Club.Player.Name) = \{First,\ Last,\ Nickname,\ \perp\}$
- $continuation(obj(\&1),\ Player.Name) = \{First,\ Last,\ Nickname\}$
- $continuation(obj(\&24),\ Club) = \{\}$

Note that in Definition 5.3 we only consider data paths originating from the object that is the first argument of the *continuation* function. By partially removing this restriction, allowing the data paths to be within the given object, and imposing a limit on the length of the simple path expression that is the second argument of the *continuation* function we arrive at the following definition.

Definition 5.4 *Let o be an object, $k \geq 1$, and let pe be a simple path expression of length n, $0 \leq n \leq k$. Then we define $continuation^k(o, pe)$ as follows.*

- *If $n = k$ then*
  - $continuation^k(o, pe) \supseteq \{l \mid \exists$ *a data path p within o, not necessarily rooted at o, that is an instance of $pe.l\}$.*
  - $continuation^k(o, pe) \supseteq \{\perp \mid \exists$ *a data path $p = o_0, l_1, o_1 \cdots l_n, o_n$, within o, that is an instance of pe and o_n is an atomic object$\}$.*
  - $continuation^k(o, pe)$ *contains nothing else.*
- *Otherwise (if $n < k$) $continuation^k(o, pe) = continuation(o, pe)$.*

Definition 5.4 translates into the following. Given an object o and a number k, we compute k-continuations for any simple path expression of length less than k as before. However, for path expressions of length precisely k we allow the data paths to be rooted at not only the given object o but also any object within o.

Example 5.3 *Consider the premiership object in Figure 5.1. The following examples illustrate Definition 5.4.*

- $continuation^1$*(premiership, Name)* = {*Official, Nickname, First, Last,* $\perp$}
- $continuation^2$*(premiership, Club)* = {*Name, Player, Stadium, Captain*}
- $continuation^2$*(premiership, Player.Name)* = {*First, Last, Nickname,* $\perp$}
- $continuation^3$*(premiership, Player.FormerClub.Player)* = {*Name, FormerClub, Nationality, Number*}

The next lemma characterizes the relationship between the functions *continuation* and $continuation^k$.

Lemma 5.2 *Let* o *be an object,* $k \geq 1$, *and* pe *a simple path expression of length* n, $0 \leq n \leq k$. *Then we have:*

- $continuation^k(o, pe) = continuation(o, pe)$ *for* $n < k$
- $continuation^k(o, pe) \supseteq continuation(o, pe)$ *for* $n = k$
- *if* $n = k$, *pe begins with* l, *where* $\langle l, id\rangle \in value(o)$, *and* l *is unique within o, then* $continuation^k(o, pe) = continuation(o, pe)$.

Proof: The first part of the lemma follows directly from Definition 5.4. The second part of the lemma follows from the fact that all data paths rooted at o are also within o. Therefore, for the same object o and simple path expression pe, the set of data paths considered in Definition 5.3 is a subset of the set of data paths considered in Definition 5.4. The third part of the lemma is a consequence of the fact that any instance data path of pe must be rooted at o because no object references within o, other than the one coming from o, has label l. Thus, in Definition 5.4 only the data paths rooted at o are effectively considered, which is the the set of data paths considered in Definition 5.3.

Example 5.4 *The following examples illustrate the three different cases in Lemma 5.2. Consider obj(&1) that is within the premiership object from Figure 5.1. Let us call this object reds. Then we have:*

- $continuation(reds,\ Player) = continuation^2(reds,\ Player) = \{Name,\ Number,\ Nationality,\ FormerClub\}$.
- $continuation^2(reds,\ Name.Nickname) = \{English,\ French,\ \perp\} \supset continuation(reds,\ Name.Nickname) = \{\perp\}$
- $continuation(reds,\ Captain.Name) = continuation^2(reds,\ Captain.Name) = \{First,\ Last,\ Nickname\}$.

5.3 Representative-Object Definitions

A *representative object* for a semistructured object o in OEM is any implementation of the *continuation* function for o. We refer to these implementations as representative *objects* because they are implemented in practice as objects in OEM. However, as discussed in later sections of this chapter, there are many different ways to represent the *continuation* function, and not all are objects in the usual sense. For instance, we discuss automaton-based representations.

In this section we define two different kinds of representative objects. First, we define the concept of a *full* representative object (FRO) for an object in OEM and justify this definition by describing how a FRO supports the motivating applications from Section 5.1.1. Then we define the concept of a *degree-k* representative object (k-RO) for an object in OEM. k-ROs are often less complex than FROs and can be used to approximate FROs. We also discuss the extent to which the motivating applications are supported by k-ROs.

5.3.1 Full Representative Objects

The *full* representative object is an implementation of the *continuation* function, restricted to a particular object. Formally.

Definition 5.5 *Let o be a semistructured object. Then the function $continuation_o(pe) = continuation(o, pe)$, where pe is a simple path expression, is a* full representative object (FRO) *for o.*

In order to justify this definition, we show how a FRO supports the motivating applications from Section 5.1.1.

Schema discovery

This application is the primary motivation for investigating representative objects. Recall that by schema discovery we mean navigating through a given object and keeping track of the labels of the object references that we traverse. By using the FRO of an object we can perform schema discovery very quickly and efficiently. We illustrate the point with an example of exploration of the premiership object in Figure 5.1. This approach to exploration has been implemented in the *DataGuide* feature of Lore, a database system using the OEM, as discussed in [GW97].

Example 5.5 *Suppose we start at the root object which is the* premiership *object. If we ask the query* $continuation_{premiership}(\epsilon)$ *we get the labels of links leading from the root. In this case, the only label is Club. The query* $continuation_{premiership}(Club)$ *then lets us see all the labels of links leading from Club objects within the premiership. These labels are Name, Player, Captain, and Stadium. Suppose we are interested in players. Then we may explore from Player by asking the query* $continuation_{premiership}(Club.Player)$, *whereupon we find that links out of Player objects can be labeled Name, Number, Nationality, or FormerClub. In the Lore DataGuide, the queries are submitted by clicking on the node we wish to expand, and after the sequence of queries described above, the presentation of (part of) the representative object would be as it appears in Figure 5.2.*

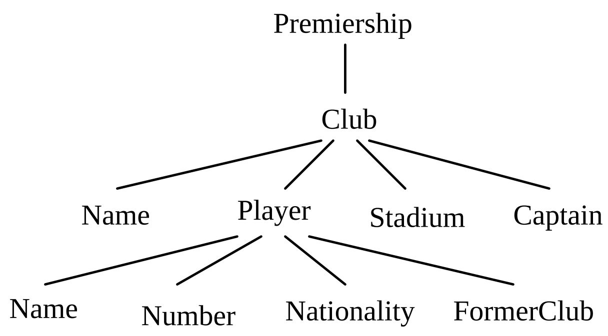

Figure 5.2: Displaying part of the FRO for the premiership object.

Path queries

Many interesting queries over semistructured data necessarily involve wild cards, because the schema of the data is not known in advance or may change often. The FROs can be used to answer efficiently such queries by finding all simple path expressions that have instance data paths within a given object and also match the wild-card pattern in a query. We illustrate the point with an example. The wild-card pattern syntax used in the example is described in [AQM+96] and the path expressions expressible in it are called *general path expressions*. In our example we use only "?", which denotes an optional label and "%", which matches any number of characters of one label.

Example 5.6 *Consider the following pattern* gpe = Club(.Player)?.(Na%) *and the premiership object in Figure 5.1. In other words, we are looking for simple path expressions that have instance data paths within the premiership object and start with* Club *followed optionally by* Player *and end with a label beginning with "Na".*

- *First we find* $continuation_{premiership}(\epsilon) = \{\text{Club}\}$.
- *The label* Club *matches the head of gpe, the tail of gpe is* (Player.)?(Na%).
- *Then we find* $continuation_{premiership}(\text{Club}) = \{\text{Name}, \text{Player}, \text{Captain}, \text{Stadium}\}$.
- *Only the label* Player *matches the head of* (Player.)?(Na%), *but because the head is an optional label, we have two simple path expressions that match gpe so far:* Club, *and* Club.Player. *The remaining tail is* Na%.
- $continuation_{premiership}(\text{Club.Player}) = \{\text{Name}, \text{Nationality}, \text{Number}, \text{FormerClub}\}$.
- *Both* Name *and* Nationality *match* Na% *so we have three simple path expressions that match gpe completely:* Club.Name, Club.Player.Name, *and* Club.Player.Nationality.

Query optimization

In order to find whether a simple path expression pe has any instance data paths originating from an object o we compute $continuation_o(pe)$. Recall that $continuation(o, pe)$ is defined to be nonempty if pe has an instance data path originating from o. Since $continuation_o(pe) = continuation(o, pe)$ then an empty result means that pe does not have any instance data paths originating from o. If the result is not empty, then pe has at least one instance data path originating from o.

5.3.2 Degree-k Representative Objects

Degree-k representative objects (k-RO) are defined similarly to full representative objects (FRO) but using the $continuation^k$ function instead of $continuation$. Formally.

Definition 5.6 *Let o be a semistructured object. Then the function $continuation_o^k(pe) = continuation^k(o, pe)$, where $k \geq 1$ and pe is a simple path expression, is a* degree-k representative object (k-RO) *for o.*

While k-ROs, in general, only approximately support the motivating applications from Section 5.1.1, they take less space (usually) than FROs and may be faster to construct. Before we show the extent to which k-ROs support the motivating applications, we describe a method of computing $continuation_o^k(pe)$, an approximation of $continuation_o(pe)$, from a k-RO.

Let o be an object, R_k a degree-k representative object for o, and $pe = l_1.l_2 \cdots l_n$ a simple path expression. We consider the following three cases.

- If we have that $n < k$ then by using R_k we can find $continuation_o^k(pe)$ and because $continuation^k(o, pe) = continuation_o(pe)$ we get the exact value of $continuation_o(pe)$.
- If $n = k$ we can find $continuation_o^k(pe)$ and by Lemma 5.2 the result is a superset of $continuation_o(pe)$.
- If we have that $n > k$ then we find $continuation_o^k(l_{n-k+1}.l_{n-k+2} \cdots l_n)$. The result is a superset of $continuation_o(pe)$. We can check if $l_{i+k} \in continuation_o^k(l_i.l_{i+1} \cdots l_{i+k-1})$ for $i = 1..n-k$. If any of these conditions does not hold then $continuation_o(pe)$ is empty and thus we have its exact value.

Consider the motivating applications from Section 5.1.1. We describe how they are supported by a k-RO, using the approximation of $continuation_o(pe)$ provided by the k-RO.

Schema discovery

As in the FRO case we start at the root object. As long as the length of the simple path expression pe that we have followed is less than k we can compute the exact value of $continuation_o(pe)$ and thus the k-RO provides the same support as a FRO. If the length of the pe is at least k then we have to use the approximation of $continuation_o(pe)$ provided by the k-RO. The consequence is that the discovered schema will contain the actual schema, but may also have some paths that do not exist within o.

The next example illustrates the difference between using a full representative object and a degree-k representative object for schema discovery.

Example 5.7 *The setup is identical to Example 5.5 with the exception that we use a* 1-*RO instead of a FRO. Again we discover the schema of the* premiership object *but using* $continuation^1$ *instead of continuation. We start at the* premiership *and ask the query* $continuation^1_{premiership}(\epsilon)$ *we get the labels of links leading from the root. Since the length of the path expression* ϵ *is 0 which is less than 1, we get* $continuation_{premiership}(\epsilon) = \{Club\}$. *The next query,* $continuation^1_{premiership}(Club)$, *in principle, can give a superset of* $continuation_{premiership}(Club)$. *However, in this case, it yields the same result, namely* $\{Name, Player, Captain, Stadium\}$. *Suppose we are interested in Names. Thus, we* would like to ask $continuation^1_{premiership}(Club.Name)$, *but this expression is not legal because the length of the path expression is 2, which is greater than the degree of the continuation function. We can only ask* $continuation^1_{premiership}(Name)$, *which yields the following result:* $\{Official, Nickname, First, Last, \perp\}$. *Note that this result contains one label that is not in the "real" result. Indeed* First *and* Last *are not in the continuation of Club.Name for the* premiership *object but is introduced because of the use of* 1-*RO instead of FRO. The situation is depicted in Figure 5.3.*

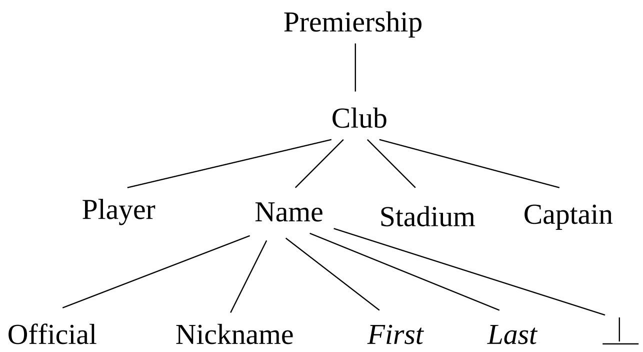

Figure 5.3: Displaying part of the 1-RO for the premiership object.

Path queries

The procedure described in the FRO case remains the same. When we compute continuations of simple path expressions of length at least k we have to use the approximation instead of the actual value. Thus, the final set of matched simple path expressions will be a superset of the actual set, and therefore each simple path expression of length at least k in the set should be verified.

Query optimization

If the approximation of the continuation of the given simple path expression pe is empty then pe has no instance data paths originating from the given object. If the result is nonempty, however, pe may or may not have instance data paths originating from the given object.

5.4 Implementation of FROs in OEM

In this section we describe one particular implementation of FROs in OEM. In fact this is how we have implemented FROs (called DataGuides) in the Lore DBMS [AQM$^+$96, GW97]. A FRO, implemented in OEM, consists of an object R_o (in OEM) and an algorithm for computing the function $continuation_o$ from R_o, where o is the represented object. By implementing FROs in OEM we gain the advantage of storing and querying the object part of the FROs in the same way as ordinary objects in OEM. We also define minimal FROs (in OEM) that allow computing the *continuation* function very efficiently.

Before we describe the implementation of FROs in OEM, we present Algorithm 5.1 that, for a given object o, computes the continuation of a simple path expression pe. The algorithm first explores o for instance data paths of pe, originating from o, in a breadth-first manner. For every such data path, only the last object in the data path is considered. Then the continuation of pe is the set of all the different labels of object references of those objects, plus $\perp$ if any of those objects is atomic.

Let o be an object and $pe = l_1.l_2 \cdots l_n$, $n \geq 0$, a simple path expression. The algorithm in Figure 5.4 computes $continuation_o(pe)$.

Then we define the implementation of FROs in OEM as follows.

Definition 5.7 *Let o_1 and o_2 be objects in OEM. Then o_1, along with Algorithm 5.1, is a* full representative object in OEM *for o_2 if for any simple path expression pe we have*

Algorithm 5.1

Input: semistructured object o in OEM
simple path expression $pe = l_1.l_2 \cdots l_n, n \geq 0$
Output: $continuation_o(pe)$

```
// Initialization
   C_0 = {o}
// Iteration
// at iteration i find all instance data paths of l_1.l_2···l_i rooted at o
   for i = 1..n do
      C_i = ∅
      for s ∈ C_{i-1} do
         for each ⟨l_i, id⟩ ∈ value(s) do
            add object(id) to C_i
         end for
      end for
      if C_i == ∅ then
      // no instance data paths for l_1.l_2···l_i rooted at o
         return ∅
   end for
// Continuation computation
   C = ∅
   for s ∈ C_n do
      if s is atomic then
         add ⊥ to C
      else
         for each ⟨l, id⟩ ∈ value(s)
            add label l to C
         end for
      end for
   return C
```

Figure 5.4: Algorithm for computing $continuation_o(pe)$ from o.

$continuation_{o_1}(pe) = continuation_{o_2}(pe)$.

From Definition 5.7 it follows that if o_1 is a FRO in OEM for o_2 then o_2 is a FRO in OEM for o_1. Also any object o is a FRO in OEM for itself.

Remark 5.3 *Formally, when we talk about FROs in OEM we always have to include Algorithm 5.1 or another algorithm that computes the continuation function from an object in OEM. In this section we only consider FROs in OEM so we will omit Algorithm 5.1 and refer to the object part as the full representative object.*

5.4.1 Minimal FROs

Form Definition 5.7 it follows that there are many FROs (in OEM) for a given object, including the object itself. Ideally, we want to choose the one that allows Algorithm 5.1 to compute the *continuation* function fastest. Each iteration of the first part Algorithm 5.1 takes time proportional to the size of C_i. Thus, the FRO for which the size of C_i at each iteration is smallest allows the fastest computation. The next definition describes a particular kind of FROs (in OEM) for which C_i always contains at most one complex object and at most one atomic object.

Theorem 5.4 *Let R_o be a FRO (in OEM) for o. Then R_o is a* minimal *FRO if any simple path expression $pe = l_1.l_2 \cdots l_n, n \geq 0$, has at most one instance data path originating from R_o and ending with a complex object and at most one instance data path originating from R_o and ending with an atomic object.*

We prove Theorem 5.4 by showing that at each iteration of the first part (breadth-first exploration) of Algorithm 5.1 for R_o C_i contains at most one complex and one atomic object for any simple path expression. Before the first iteration, the size of C_0 is 1. Thus, for a simple path expression of length 0 (ϵ) the assertion holds since the iteration part of Algorithm 5.1 is not executed. Let $pe = l_1.l_2 \cdots l_n, n \geq 1$ be a simple path expression. Let $n \geq k > 0$ be the smallest k for which after the kth iteration C_k contains more than one atomic object or more than one complex object. Then we can construct at least two data paths that are instances of the same simple path expression and end with objects of the same kind (atomic or complex). Let the sole complex object in C_i after the ith iteration be o_i, for $i = 1 \cdots k-1$. At the kth iteration C_k contains at least two different objects o_k and

o'_k of the same kind. Consider the data paths $R_o, l_1 \cdots o_{k-1}, l_k, o_k$ and $R_o, l_1 \cdots o_{k-1}, l_k, o'_k$. Both data paths originate from R_o, end with objects of the same kind, and are instances of the simple path expression $l_1.l_2 \cdots l_k$. So, we have a contradiction of Definition 5.4, and thus the assertion holds in all cases.

We will use the assertion proved above to calculate the running time of Algorithm 5.1 for a minimal FRO (in OEM) R_o for o and a simple path expression pe of length n. The number of iterations of the first part of Algorithm 5.1 for R_o is n. The size of S before each iteration is at most 2. Thus, if we can retrieve the object references that have a particular label for a given object in constant time then each iteration takes constant time. The second part of Algorithm 5.1 takes time proportional to the size of $continuation_o(pe)$. Thus, the computation of the continuation of a simple path expression for an object given a minimal FRO (in OEM) for this object takes linear time with respect to the length of the simple path expression and the number of different labels in the computed continuation.

5.5 Construction of Minimal FROs

In this section we present a method for constructing minimal FROs in OEM. The method consists of three major steps: construction of a nondeterministic finite automaton (NFA) from a given object, determinization of this NFA and minimization of the resulting deterministic finite automaton (DFA) that yields another DFA, and construction of a minimal FRO from this DFA. We also prove the correctness of this method.

5.5.1 Finite automata

Finite automata are used in many areas of computer science and are studied extensively. A detailed study is given in [HU79]. A finite automaton $(Q, \Sigma, \delta, q_0, F)$ consists of a finite set of states Q, a finite alphabet Σ, and transitions from one state to another on a letter of the alphabet ($\delta : Q \times \Sigma \mapsto Q$). One state, q_0, is designated as the start state and there are one or more end (accepting) states F. All the words formed by the sequences of letters on transitions from the start state to an end state form the language accepted by the automaton.

5.5.2 Construction of a NFA from an object in OEM

Every object in OEM can be viewed as a NFA in a straightforward manner. The objects correspond to states and the object references and their labels correspond to transitions and their respective letters. Before we show formally how we construct the NFA corresponding to an object o in OEM, we introduce the function *state* that maps every object within o to a unique automaton state corresponding to it. We extend this function to map a set of objects within o to the set of the automaton states corresponding to them. We also define the following terms that characterize the object o in OEM. Let $\mathcal{A}$ be the set of all atomic objects within o, $\mathcal{C}$ the set of all complex objects within o, and $\mathcal{D}$ the set of all objects within o. Note that $\mathcal{D} = \mathcal{A} \cup \mathcal{C}$. Let also $\mathcal{L}$ be the set of all different labels of object references within o. The NFA $(Q, \Sigma, \delta, q_0, F)$ corresponding to o is constructed as follows.

- $Q = state(\mathcal{D}) \cup \{end\}$
- $\Sigma = \mathcal{L} \cup \{\perp\}$
- $\delta(state(c), l) = state(object(id))$ for $\forall c \in \mathcal{C}$ and $\forall \langle l, id \rangle \in value(c)$
- $\delta(state(a), \perp) = end$ for $\forall a \in \mathcal{A}$
- $q_0 = state(o)$
- $F = Q$

The next example illustrates the construction of a NFA from an object in OEM.

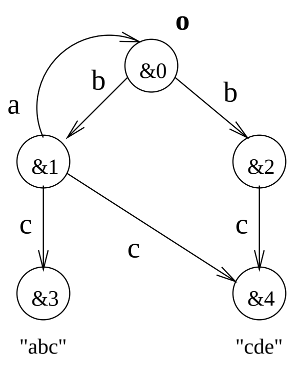

Figure 5.5: Graph representation of the example object o.

Example 5.8 *Consider the object o represented as a graph in Figure 5.5. The NFA constructed from o is shown in Figure. 5.6. The starting state q_0 corresponds to o, i.e., $q_0 = state(o)$. We also have $q_i = state(o_i)$ for $i = 1..4$. All six states are accepting. Since o_3 and o_4 are atomic objects we have transitions on $\perp$ from their corresponding states, q_3 and q_4, to end. The rest of the transitions correspond to the object references within o.*

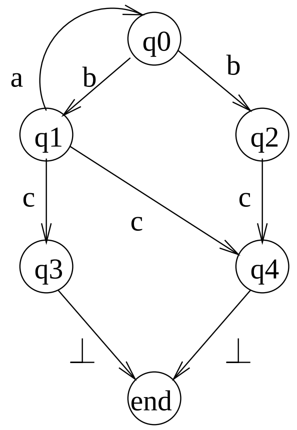

Figure 5.6: A NFA constructed from the example object o.

5.5.3 Determinization and minimization of a NFA

The determinization (conversion to a DFA) and minimization of a NFA is a very well studied problem. The determinization of a NFA can take exponential time with respect to its number of states [HU79]. If, however, the NFA has a tree structure, i.e, every state has only one incoming transition and there are no cycles, then the determinization takes linear time. The best algorithm for minimization of a DFA takes $n \log n$ time where n is the number of states of the DFA [Hop71].

5.5.4 Construction of a minimal FRO from a DFA

The transformation from a DFA to an object in OEM is straightforward except for the treatment of some states with which we associate two different objects, one atomic and one complex. With the rest of the states we associate a unique object. We also associate an object reference with each letter transition. Before we formally describe the construction of a minimal FRO from the DFA $(Q, \Sigma, \delta, Q_0, F)$ we introduce two functions, *atomic_obj* that maps a state to its corresponding atomic object (if any) and *complex_obj* that maps a state to its corresponding complex object (if any). The minimal FRO corresponding to the DFA is constructed as follows.

- Let $S_a = \{q \mid q \in Q, \delta(q, \perp) = end\}$.
- Let $S_c = \{q \mid q \in Q, \exists l, r$ such that $l \in \Sigma, l \neq \perp, r \in Q$ and $\delta(q,l) = r\}$, i.e. S_c is all states with a non-$\perp$ transition out.
- For $\forall q \in S_a$ $atomic\_obj(q)$ is a unique atomic object.
- For $\forall q \in S_c$ $complex\_obj(q)$ is a unique complex object and $value(complex\_obj(q)) = \{\langle identifier(atomic\_obj(p)), l\rangle \mid \delta(q,l) = p$ and $atomic\_obj(q)$ is defined$\} \cup \{\langle identifier(complex_obj(p)), l\rangle \mid \delta(q,l) = p$ and $complex\_obj(q)$ is defined$\}$.
- If $complex\_obj(Q_o)$ is defined then the minimal FRO, R_o, is $complex\_obj(Q_o)$. Otherwise $R_o = atomic\_obj(Q_o)$.

Example 5.9 *The DFA that we get after the determinization and minimization of the NFA in Figure 5.6 is shown in Figure 5.7.*

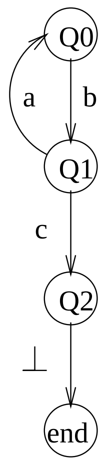

Figure 5.7: A minimized DFA.

We construct the minimal FRO in OEM for o, shown in Figure 5.8, from this DFA by creating the following three objects.

- $R_o = object(\&10) = complex\_obj(Q_0), value(R_0) = \{\langle \&11, B\rangle\}$.
- $object(\&11) = complex\_obj(Q_1), value(object(\&11)) = \{\langle \&10, A\rangle, \langle \&12, C\rangle\}$.

- $object(\&12) = atomic\_obj(Q_2)$.

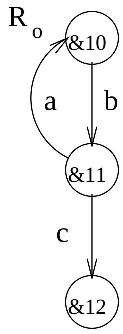

Figure 5.8: A minimal FRO constructed from a DFA.

Example 5.10 *As an illustration of the method described in this section, Figure 5.9 shows the minimal FRO in OEM for the premiership object in Figure 5.1. Note that there are two links labeled "Name" coming from the same "Club" object. Having two links with the same label from the same object does not contradict Definition 5.4, because one of the "Name" subobjects is atomic and the other one is complex.*

5.5.5 Correctness proof

In order to prove that the method we present is correct we have to show that the object constructed in the third step of the method is indeed a minimal FRO in OEM for the original object.

Let o be an object, N_o the NFA constructed from o as described in Section 5.5.2, D_o the DFA obtained after the determinization and minimization of N_o, and R_o the object constructed from D_o as described in Section 5.5.4. We will show that $continuation_o(pe) = continuation_{R_o}(pe)$ for any simple path expression pe by showing that $continuation_o(pe) \subseteq continuation_{R_o}(pe)$ and $continuation_o(pe) \supseteq continuation_{R_o}(pe)$.

Let $pe = l_1.l_2 \cdots l_n$, $n \geq 0$, be a simple path expression and $l \in continuation_o(pe)$. Then pe has an instance data path $p = o, l_1, o_1, l_2 \cdots o_n$. From the construction of N_o we have:

- $\delta(state(o), l_1) = state(o_1)$.

- $\delta(state(o_{i-1}), l_i) = state(o_i)$, for $i = 2 \cdots n$.

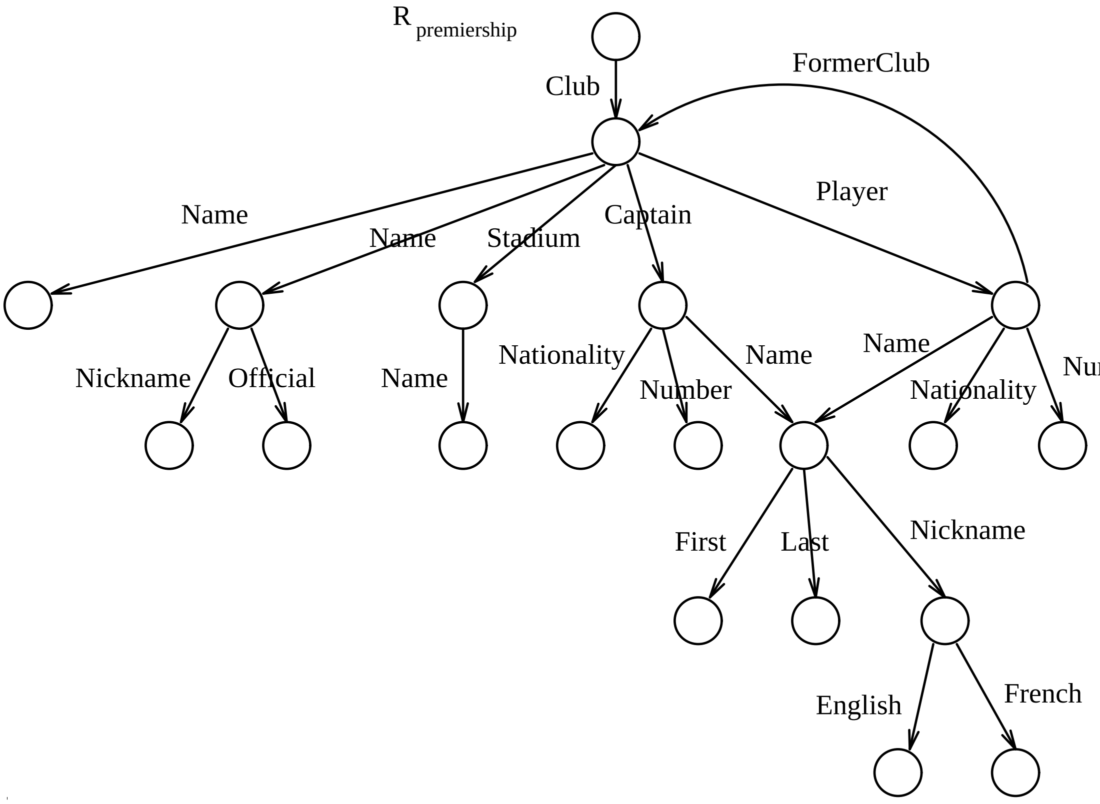

Figure 5.9: The minimal FRO for the premiership object.

There are two possible cases for l, $l = \perp$ and $l \neq \perp$. In the first case, $l = \perp$, we have that o_n is atomic and thus $\delta(state(o_n), \perp) =$ end. In the second case, $l \neq \perp$ we have that o_n has an object reference to an object o_{n+1} labeled with l and thus $\delta(state(o_n), l) = state(o_{n+1})$. Therefore, in both cases the word $l_1 l_2 \cdots l_n l$ is accepted by N_o. The DFA D_o is equivalent to N_o by construction and therefore D_o and N_o accept the same language. Thus, the word $l_1 l_2 \cdots l_n l$ is accepted by D_o. Then there are states Q_i in D_o for $i = 0..n+1$, such that $\delta(Q_{i-1}, l_i) = Q_i$ for $i = 1..n$, $\delta(Q_n, l) = Q_{n+1}$, Q_0 is the start state of D_o, and Q_{n+1} is an accepting state. Then from the construction of R_o we have that $\langle identifier(complex\_obj(Q_i)), l_i \rangle \in value(complex\_obj(Q_{i-1}))$ for $i = 1..n-1$. Thus, the data path $P = R_o, l_1 \cdots l_{n-1}, complex\_obj(Q_{n-1})$ exists. If $l = \perp$ we have that $Q_n \in S_a$ and thus, $atomic\_obj(Q_n)$ is defined. Therefore, $\perp \in continuation_{R_o}(pe)$ because of the data path $P, l_n, atomic\_obj(Q_n)$. If $l \neq \perp$ we have that $Q_n \in S_c$ and thus $complex\_obj(Q_n)$ is defined. Therefore, $l \in continuation_{R_o}(pe)$ because of the data path

$P, l_n, complex\_obj(Q_n), l, obj$ where obj is either $complex\_obj(Q_{n+1})$ or $atomic\_obj(Q_{n+1})$, whichever is defined. Therefore we proved that $continuation_o(pe) \subseteq continuation_{R_o}(pe)$.

Similarly we can show that if $l \in continuation_{R_o}(pe)$ then the word $l_1 l_2 \cdots l_n l$ is accepted by D_o and thus by N_o. Then we can show that $l \in continuation_o(pe)$ and therefore $continuation_o(pe) \supseteq continuation_{R_o}(pe)$. We can also show that R_o is a minimal FRO from its construction from D_o and the fact that D_o is a DFA. With this we conclude the proof of correctness of the minimal-FRO construction method.

5.6 Construction of a 1-RO

The simplest representative object to construct is the 1-RO. While the 1-RO only guarantees that its paths of length 2 exist within the represented object, it nonetheless indicates the set of possible labels that may succeed an individual label. Furthermore, the 1-RO provides a very compact description of the represented object, is easy to construct, and easy to comprehend. We can represent the 1-RO as a graph with the nodes corresponding to labels. Intuitively, the 1-RO contains each unique label exactly once, and contains an edge between two labels if the simple path expression consisting of the two labels has an instance data path within the given object. For example, Figure 5.11 shows the 1-RO for the semistructured object s shown in Figure 5.10. In this section, we describe an algorithm for constructing the 1-RO for an object in OEM in one physical, sequential scan of all objects within the given object.

5.6.1 1-RO Algorithm

The goal is to find all pairs of labels (l_1, l_2) such that there is a data path o_0, l_1, o_1, l_2, o_2 within the given object. Each object o_i contains pairs of identifiers and labels (object references) but does not contain the labels of incoming links. Thus we must must examine all objects that have links to o_i. For all complex objects we remember all triples $(identifier(o_i), l_{i+1}, identifier(o_{i+1}))$ in the *id* table. For the root object r we remember a special triple $(null, \epsilon, identifier(r))$. For each atomic object a we remember a special triple $(identifier(a), \perp, null)$. Then we join the id table with itself on $identifer(o_{i+1}) = identifer(o_i)$ to produce (l_{i+1}, l_{i+2}) pairs. Note that the *null* values *do not* join.

The id table can be built in one scan of the objects (in any order). The cost of computing the pairs of labels then depends on the size of the id table. If it fits in memory, then an

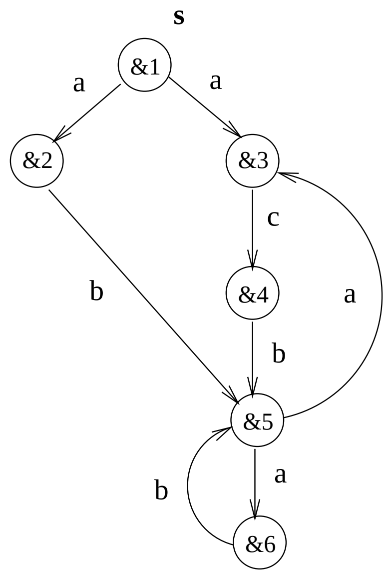

Figure 5.10: An example semistructured object s.

in-memory join is performed for no extra I/O cost. Otherwise, the additional I/O cost is that of a join, which is $2 * size(id\ table)$ for a two-pass hash (self-)join.

The result table, called *label* table, is then indexed by l_1 so that lookups are efficient. Duplicate label pairs are discarded. The id table and the label table for the object in Figure 5.10 are shown in Table 5.1.

5.6.2 Computing 1-continuations

Using the 1-RO we can find the continuation of a simple path expression consisting of a single label l or ϵ, the path expression of length 0. We lookup all pairs (l, l_2) in the label table; the set of all such l_2 is the 1-continuation of l. The time required is the cost of an index lookup: $O(1)$ if the label table index fits in memory and nothing if the label table itself fits in memory.

5.7 Construction of k-ROs

In this section, we present a construction method for k-representative objects based on finite automata. We treat simple path expressions as strings over the alphabet of OEM labels. Consider the set of simple path expressions of length up to $k+1$ that have instance data

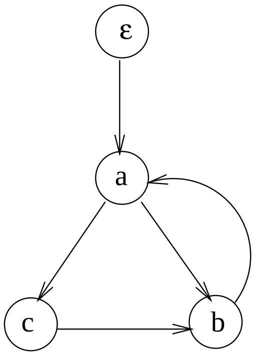

Figure 5.11: Graph representation of the 1-RO for the semistructured object s.

identifier1	*label*	*identifier2*
nil	ϵ	&1
&1	a	&2
&1	a	&3
&2	b	&5
&3	c	&4
&4	b	&5
&5	a	&3
&5	a	&6
&6	b	&5

label1	*label2*
ϵ	a
a	b
b	a
c	b
a	c

Table 5.1: The id and label tables for the semistructured object s.

paths within the represented object. The automaton that accepts the language represented by this set serves as a k-RO.

5.7.1 Constructing an automaton that serves as a k-RO

We assume that we have computed the set P of all simple path expressions of length up to $k+1$ that appear in the object o being represented. Consider an alphabet V consisting of the labels in o. Then P represents a finite, and hence regular, language over the alphabet V. Using standard techniques [HU79], we construct a finite automaton A that recognizes the language P. (We assume that the automaton A is minimized using the subset construction method [HU79].)

Consider the semistructured object shown in Figure 5.10. Table 5.2 shows the 3-paths of

$\epsilon.\epsilon.a$
$\epsilon.a.b$
$\epsilon.a.c$
$a.c.b$
$a.b.a$
$b.a.b$
$b.a.c$
$c.b.a$

Table 5.2: Simple path expressions of length 3 with instance data paths within object s.

that object, that is, the set P above for $k = 2$. A finite automaton that accepts the language suggested by P (interpreting simple path expressions as strings) is shown in Figure 5.12. The initial state is marked with a short arrow, and the accepting states are circled.

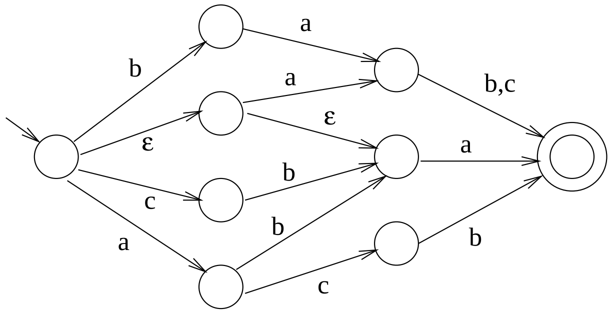

Figure 5.12: Finite-automaton-based 2-RO for the example object.

5.7.2 Computing k-continuations

Having an automaton-based representation of the k-representative object as described above allows us to compute k-continuation as follows. Suppose we wish to know the continuation of the simple path expression $l_1.l_2 \cdots l_k$. We start in the initial state of the automaton and follow the transition with label l_i for $i = 1 \cdots k$ to reach a state s_k. (If, at some stage, we are in a state with no transition with the desired label, the continuation is empty.) Let A be the set of transitions that go from s_k to an accepting state. The set of labels in A is the continuation of the given simple path expression.

If we use an index to represent the transitions out of each state in the automaton, finding the next state requires at most $O(logl)$ time, where l is the number of labels in the represented object. Finding the state s_k therefore requires at most $O(klogl)$ time. If there are c labels in the continuation of the given simple path expression, we can retrieve the labels on the transitions out of s_k in no more than $O(c)$ time. (Note that all these transitions must lead to accepting states, since every path of length $k+1$ in the automaton leads to an accepting state.) Thus, the total time required to compute the k-continuation is $O(klogl + c)$. In practice, we can achieve a running time close to $O(k + c)$, so the time required to find the k-continuation is bounded by $O(klogl + c)$.

5.8 Concluding Remarks

In this chapter we have introduced the representative object concept that provides a concise representation of the inherent schema of a semistructured data source. We made the case that representative objects are very useful for semistructured data and show some of their primary uses in data browsing, query formulation, and query optimization. We also described an implementation of full representative objects (FROs) in OEM that has the advantage that the data part of the FRO can be stored and queried as an object in OEM. We presented a construction method for an important class of FROs: minimal FROs. Minimal FROs allow efficient querying of the schema of the represented data. Since constructing minimal FROs may potentially have very high complexity we described several alternative approaches to constructing k-ROs that are approximations of an FRO. In many case, even a 1-RO provides a good approximation of an FRO.

Chapter 6

Extracting Schema From Semistructured Data

6.1 Introduction

In this chapter, we present an approach to discovering an approximate schema for semistructured data using the greatest fixpoint semantics of monadic datalog programs. This approach differs from the representative objects, described in Chapter 5, in two major ways: the form of the mined structure and the assumption of data noise. The approach presented in this chapter discovers structure in the form of a datalog program, whereas representative objects summarize structure in the form of objects expressed in the same data model as the mined data. Furthermore, representative objects assume that the given semistructured data is perfect, in the sense that there are no unintended irregularities that may result from human or integration mistakes. On the other hand, the approach presented in this chapter allows for data noise.

Note that the implicit structure in a particular semistructured data set may be of varying regularity. Indeed, we should not expect in general to be able to type a data set perfectly. The size of a *perfect* typing (a notion that we will study) may be quite large, e.g., be roughly of the order of the size of the data set, which would prohibit its use for query optimization and render it impractical for graphical query interfaces. Thus, we consider *approximate* typings, i.e., an object does not have to fit its type definition precisely, but on average, objects should be "close". We study the trade-off between the quality of a typing and its compactness. More precisely, the typing problem and its trade-off can be formulated as follows. Suppose we have selected a type description language and a measure for type sizes, as well as a distance function over data sets. The problem then is: given a data set I, find a typing τ and a data set J of typing τ, such that the size of τ is smaller than a certain threshold, and the distance between I and J is minimized. In other words, we want to find a τ that is small enough and such that I presents as few inconsistencies as possible with respect to τ. (The dual problem is the minimization of the size of τ for a given upper bound on the distance between I and J.)

The first key issue is the choice of a description language for types. Our typing is inspired by the typing found in object databases [Cat94], although it is more general since we allow objects to live in many incomparable classes, i.e., have multiple roles [ABGO93]. This aspect is also a clear departure from previously proposed typings for semistructured data [GW97, NUWC97, BDFS97, Suc96]. We believe, backed by our experiments, that multiple roles are essential when the data is fairly irregular. We define a typing in terms of a monadic datalog program. The intensional relations correspond to object classes, and the rules describe the inner structure of objects in classes. The greatest fixpoint semantics of the program defines the class extents.

We first consider the issue of computing perfect typings. The basis is an obvious perfect typing that consists of having one class for each distinct object. This typing is transformed into the initial monadic datalog program. A run of this program on the data set will naturally group similar objects and thus provide a (possibly much) coarser classification of objects that yields a (possibly much) more compact perfect typing. In fact, this typing is the coarsest possible if we insist on exact fit.

This perfect typing may still be much too large, unless the data is extremely regular. Therefore, we present a technique for computing an approximate typing of an appropriate size. Our technique allows the data set to be imperfect with respect to the typing. It may present extra information (edges not required by the typing are present in objects) or lack some information (edges required by the typing are missing in some objects). We will see that extra edges are easily handled with a greatest fixpoint approach, whereas missing edges are much more difficult to deal with. The crux of our technique is to merge similar classes so as to decrease the size of the typing. To this end, we employ a clustering algorithm [KPR98, Hoc82] on the classes. Intuitively, until the typing is of acceptable size (for some application-dependent notion of "acceptable size"), we perform class merges that introduce a minimal error. We consider various optimization strategies to compute this approximation and issues such as the choice of the distance function.

For a concrete example, Figure 6.1 shows the approximate typing produced by our method for the DBG dataset consisting of information about the members of the Data Base Group at Stanford. The exact notation will be explained later in this chapter. For this example, it suffices to say that each label with an arrow and superscript corresponds to a link to or from a type. This typing has only 6 types and provides very good summary of the actual contents of the DBG dataset. In contrast, the perfect typing for this dataset consists of 53 different types. Note that we have also given the intuitive meaning before each of the 6 types.

In contrast to our work, previous proposals on typing semistructured data [GW97, NUWC97, BDFS97] have focused on perfect typing and implicitly assumed that each object has a unique role. We have already mentioned some motivation for approximate typing and will later discuss further motivations for multiple roles.

We also present some some experimental results. The focus of our experiments is the quality of the typing results rather than the time performance.

The rest of this chapter is organized as follows. Section 6.2 introduces our notation and provides the intuition for choosing the specific form of typing. Section 6.3 gives a summary of our method for extracting the typing from the data. Section 6.4 deals with perfect typing and Section 6.5 with the issue of computing an approximate typing. Section 6.6 addresses recasting the original data within the approximate typing. Section 6.7 provides some experimental results. Section 6.8 concludes this chapter.

6.2 The Typing

In this chapter, we model semistructured data somewhat differently than in the previous two chapters. The underlying model is still based on directed labeled graphs, as detailed in Section 4.2.1, but we represent the information symbolically as the following two relations:

link(FromObj, ToObj, Label): Relation *link* contains all the edge information. Precisely, $link(o_1, o_2, \ell)$ corresponds to an edge labeled ℓ from object o_1 to o_2. Note that there may be more that one edge from o_1 to o_2, but, in our model, for a particular ℓ, there is at most one such edge labeled ℓ.

$$
\begin{array}{rcll}
\textit{project}: & \tau_1 & = & \overleftarrow{Project}^{3}, \overleftarrow{Project}^{4}, \overleftarrow{Project}^{5}, \overrightarrow{Project\_Member}^{3}, \overrightarrow{Name}^{0}, \\
 & & & \overrightarrow{Project\_Member}^{4}, \overrightarrow{Homepage}^{0} \\
\textit{publication}: & \tau_2 & = & \overleftarrow{Publication}^{3}, \overleftarrow{Publication}^{5}, \overrightarrow{Author}^{3}, \overrightarrow{Name}^{0}, \\
 & & & \overrightarrow{Conference}^{0}, \overrightarrow{Postscript}^{0} \\
\textit{db-person}: & \tau_3 & = & \overleftarrow{Project\_Member}^{1}, \overleftarrow{Group\_Member}^{5}, \overrightarrow{Years\_At\_Stanford}^{0}, \\
 & & & \overrightarrow{Project}^{1}, \overrightarrow{Birthday}^{5}, \overrightarrow{Degree}^{6}, \overrightarrow{Email}^{0}, \overrightarrow{Home\_Page}^{0}, \\
 & & & \overrightarrow{Original\_Home}^{0}, \overrightarrow{Personal\_Interest}^{0}, \overrightarrow{Research\_Interest}^{0}, \\
 & & & \overrightarrow{Title}^{0}, \overrightarrow{Name}^{0} \\
\textit{student}: & \tau_4 & = & \overleftarrow{Project\_Member}^{1}, \overleftarrow{Student}^{4}, \overleftarrow{Group\_Member}^{5}, \overrightarrow{Project}^{1}, \\
 & & & \overrightarrow{Advisor}^{4}, \overrightarrow{Email}^{0}, \overrightarrow{Title}^{0}, \overrightarrow{Home\_Page}^{0}, \overrightarrow{Name}^{0}, \overrightarrow{Nickname}^{0} \\
\textit{birthday}: & \tau_5 & = & \overleftarrow{Birthday}^{3}, \overrightarrow{Namex}^{0}, \overrightarrow{Month}^{0}, \overrightarrow{Day}^{0}, \overrightarrow{Year}^{0} \\
\textit{degree}: & \tau_6 & = & \overleftarrow{Degree}^{3}, \overrightarrow{Major}^{0}, \overrightarrow{School}^{0}, \overrightarrow{Name}^{0}, \overrightarrow{Year}^{0}
\end{array}
$$

Figure 6.1: Optimal typing program for DBG data set.

atomic(Obj,Value): This relation contains value information. The fact $atomic(o, v)$ corresponds to object o being atomic and having value v.

We also require that (i) each atomic object has exactly one value, i.e. *Obj* is a key in relation *atomic*, and (ii) each atomic object has no outgoing edges, i.e., the first projections of *link* and *atomic* are disjoint.

The *link* and *atomic* relations are related to the Object Exchange Model, defined in 4.2.1 as follows:

- $link(o_1, o_2, \ell)$ iff o_1 is a complex object and $\langle \ell, o_2 \rangle \in value(o_1)$.
- $atomic(o, v)$ iff o is an atomic object and $v = value(o)$.

In the following, we consider that the data comes in as an instance over *link* and *atomic* satisfying these two restrictions. We use the term *database* here for such a data set. An example of a database is given in Figure 6.2. The same database is shown as a directed labeled graph in Figure 6.3. Note that this data happens to be very regular.

FromObj	*ToObj*	*Label*
g	m	is-manager-of
j	a	is-manager-of
m	g	is-managed-by
a	j	is-managed-by
g	gn	name
j	jn	name
m	mn	name
a	an	name

Obj	*Value*
gn	"Gates"
jn	"Jobs"
mn	"Microsoft"
an	"Apple"

Figure 6.2: The *link* and *atomic* relations.

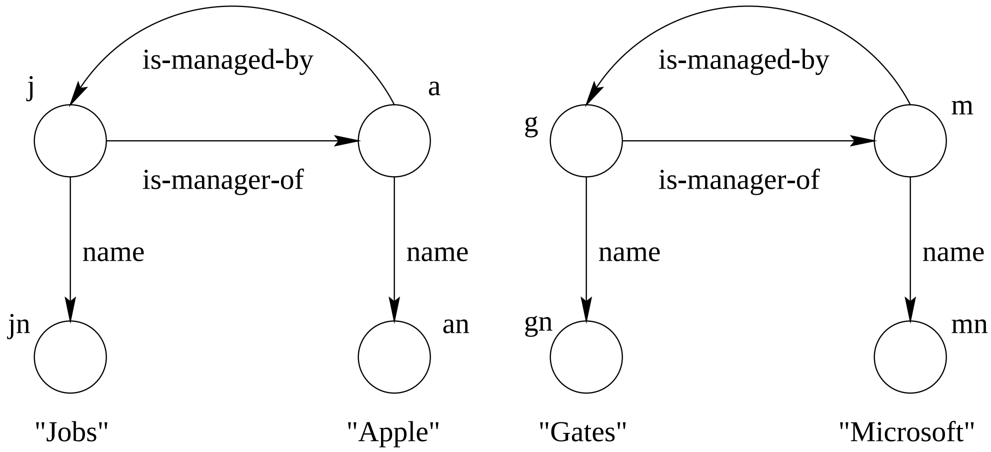

Figure 6.3: Example database as a graph.

In this chapter, we consider that a typing is specified by a datalog program of a specific form (to be described shortly). The only two extensional relations (EDB's) of the typing program are *link* and *atomic*. The intensional relations (IDB's) are all monadic and correspond to the various types defined by the program. For instance, we can consider the following typing program $\mathcal{P}_0$ for the database of Figure 6.2:

$$\begin{array}{lll} person(X) & \text{:-} & link(X,Y,\textit{is-manager-of})\ \&\ company(Y)\ \& \\ & & link(X,Y',name)\ \&\ atomic(Y',Z) \\ company(X) & \text{:-} & link(X,Y,\textit{is-managed-by})\ \&\ person(Y)\ \& \\ & & link(X,Y',name)\ \&\ atomic(Y',Z) \end{array}$$

The intuition is that g, j are persons, m, a are companies and the other objects are atomic.

6.2.1 Syntax

Typing programs are more precisely defined as follows. The extensional database (EDB) relations are *link* and *atomic*. The intentional database (IDB) relations are all monadic. Furthermore, each IDB relation is defined by a single rule of the form:

$$c(X) \ \ \text{:-} \ \ A_1 \ \& \ ... \ \& \ A_p$$

for some p, where the A_i, called the typed links, are defined as follows. Each *typed link* has one of the following forms:

1. $link(Y, X, \ell)$ & $c'(Y)$
2. $link(X, Y, \ell)$ & $c'(Y)$
3. $link(X, Y, \ell)$ & $atomic(Y, Z)$

where ℓ is some constant (a label), X is the variable in the head of the rule and Y, Z are variables not occurring in any other typed link of the rule. We will discuss some limitations introduced by this typing further on.

6.2.2 Notation

Suppose that the types (IDB's) of the program are $type_1 \cdots type_n$. Think of the atomic objects as belonging to $type_0$. The following notation for typed links greatly simplifies our presentation and will be used throughout the chapter:

- $link(Y, X, c)$ & $type_j(Y)$ is denoted by $\overleftarrow{c}^j$.
- $link(X, Y, c)$ & $type_j(Y)$ is denoted by $\overrightarrow{c}^j$.
- $link(X, Y, c)$ & $atomic(Y, Z)$ is denoted by $\overrightarrow{c}^0$.

The direction of the arrow over the label denotes whether the edge is incoming (left) or outgoing (right). The superscript denotes the type of the object at the other end of the edge.

6.2.3 Semantics

The semantics of a datalog program $\mathcal{P}$ of the form described above for a database D is defined as the *greatest fixpoint* of $\mathcal{P}$ for D. (See, e.g., [Apt91].) More precisely, let M be an instance over the schema of $\mathcal{P}$ such that M coincides with D on $\{link, atomic\}$. Then M is a *fixpoint of* $\mathcal{P}$ *for* D, if for each IDB c, $\mathcal{P}(M)(c) = M(c)$. It is the *greatest* fixpoint, if it contains any other fixpoint of $\mathcal{P}$ *for* D.

Note that this definition is correct because for a given database D and a datalog program $\mathcal{P}$, there is unique greatest fixpoint of $\mathcal{P}$ for D [Apt91]. For the data of Figure 6.2, and for the program $\mathcal{P}_0$, the greatest fixpoint is $\{person(g), person(j), company(a), company(m)\}$, which is as expected. Note that for this program, a least fixpoint semantics would fail to classify any object. The intuition behind the choice of the greatest fixpoint semantics is that we want to classify consistently as many objects as possible. The choice of a fixpoint indicates that the type of an object is justified by the types of objects connected to it.

Justifications of the above definition are also as follows. First, consider some relational data represented with *link* and *atomic* in the natural way: the entries of the tables are represented by atomic objects, the tuples by complex objects, and the labels are the attributes of relations. Consider the typing program corresponding to this schema also in a natural manner: one type is used for each relation. Then the previous typing would correctly classify the tuples. (We assume that no two relations have the same set of attributes for, in that case, their tuples would become indistinguishable.) Observe that for relational data, (i) the typing program is not recursive and thus the greatest fixpoint and the least fixpoint coincide and (ii) the data graph is bipartite in the sense that edges only go from complex objects to atomic ones.

For a second justification, consider some ODMG data [Cat94] (ignoring collections such as lists or bags that are beyond our framework). For the natural representation of this data with *link* and *atomic*, the natural typing program would correctly classify the objects.

Observe that the typed links allow us to describe locally the structure of the objects in a class c. With typed links, one can state that there is some edge labeled ℓ going to (coming from) an object in some other class c' or going to (coming from) an atomic object. Note also that the language is quite restricted. For instance, it is straightforward to see that the typing rules can be expressed in first-order logic with 2 variables (FO^2) which is a very restricted subset of first-order logic. For instance, the rule for *person* can be rewritten equivalently as:

$$\begin{aligned} person(X) \quad \Leftrightarrow \quad & \exists\, Y(link(X,Y,\textit{is-manager-of}) \ \wedge \ company(Y)) \ \wedge \\ & \exists\, Y(link(X,Y,name) \ \wedge \ \exists X(atomic(Y,X))) \end{aligned}$$

that uses only two distinct variables. The fact that we limit ourselves to a framework such as the FO^2 logic may be an asset, since that logic has nice properties [BGG97]; e.g., satisfaction is decidable for FO^2. On the other hand, observe that there is natural "typing" information that falls outside our scope. For instance, one cannot express in FO^2 some simple restrictions on the cardinality of certain kinds of links, e.g., that companies have a unique name, and therefore such restrictions cannot be expressed in our rule language. Also, observe that even some rules that use only two variables are not allowed in our typing programs, e.g., the rule

$$\begin{aligned} person(X) \quad \text{:-} \quad & link(X,Y,\textit{is-manager-of}) \ \&\ company(Y) \ \&\ link(X,Y',name) \ \& \\ & link(Y,X,\textit{is-managed-by}) \ \&\ atomic(Y',Z) \end{aligned}$$

can be expressed using two variables only, as shown below, but is outside our framework.

$$\begin{aligned} person(X) \quad \Leftrightarrow \quad & \exists\, Y(link(X,Y,\textit{is-manager-of}) \ \wedge \ link(Y,X,\textit{is-managed-by}) \ \wedge \\ & company(Y)) \ \wedge \ \exists\, Y(link(X,Y,name) \ \wedge \ \exists X(atomic(Y,X))) \end{aligned}$$

Clearly, one could consider richer typing languages, and in particular, unrestricted monadic datalog programs. In the present chapter, we focus on the previously defined simple types based on typed links.

Note that in this presentation we ignore the value of the atomic objects and assign all of them to the same type. In practice, however, it is often easy to separate the atomic values into different sorts, e.g., integer, string, gif, wav, etc. Indeed, one can also apply (application specific) analysis techniques to enrich the world of atomic types with domains such as names, dates or addresses. It is straightforward to extend the framework to handle multiple atomic types.

A more difficult extension to our framework would be to consider some apriori knowledge of the typing. Such apriori knowledge often exists in practice, e.g. when we integrate data with a known structure to semistructured data discovered on the net.

Finally, one may want to use in the typing, specific atomic values or ranges of atomic values. For instance, we can classify differently objects with values "Male" or "Female" in a *gender* subobject. These are interesting extensions that should be considered in future work.

6.2.4 Defect: Excess and Deficit

In the case of relational and object data that are very regular and with the proper typing program, we obtain a *perfect* classification of the objects. In general, one should not expect such precision. Suppose we have a program $\mathcal{P}$ that proposes a typing for a database D. We need a measure of how well $\mathcal{P}$ types D.

A first measure is the number of ground facts in D that are not used to validate the type of any object. We call this measure the *excess*, since it captures the number of facts that are in excess. More precisely, let M be the greatest fixpoint of $\mathcal{P}$ for D. A ground fact $link(o, o', \ell)$ in D is in excess if there exist no classes c and c', such that o is in $M(c)$, o' in $M(c')$ and the definition of c or c' stipulates that there is an ℓ-link from c to c'. The number of such ground facts is the *excess*.

Excess is rather easy to capture with our datalog programs and the greatest fixpoint semantics. The deficit, i.e., some information that may be missing, is much less so. To define the deficit, we need also to be given a typing assignment τ in addition to a program P and a database D, that associates a set of objects to each type. The *deficit* of τ is the minimum number of ground facts that must be added to D (invented) in order to make all type derivations in τ possible. (A subtlety is that τ does not have to be a typing since the addition of these facts may bring some objects to more classes than specified by τ.)

Example 6.1 *Suppose we are given the database shown in Figure 6.4 and the following typing program:*

$$
\begin{aligned}
type_1 &= \overrightarrow{a}^2 \\
type_2 &= \overleftarrow{a}^1, \overrightarrow{b}^0, \overrightarrow{c}^0 \\
type_3 &= \overrightarrow{b}^0, \overrightarrow{d}^0
\end{aligned}
$$

Consider two type assignments, τ_1 and τ_2, that both map o_i to $type_i$, for $i = 1, 2, 3$. They differ in that τ_1 maps o_4 to $type_2$ and τ_2 maps o_4 to $type_3$. Then the defect of τ_1 with respect to the given database and program is 2 because we have to "invent" one base fact, namely $link(o_1, a, o_4)$ and have to disregard $link(o_4, d, o_0)$. Thus, the excess is 1 and the deficit is

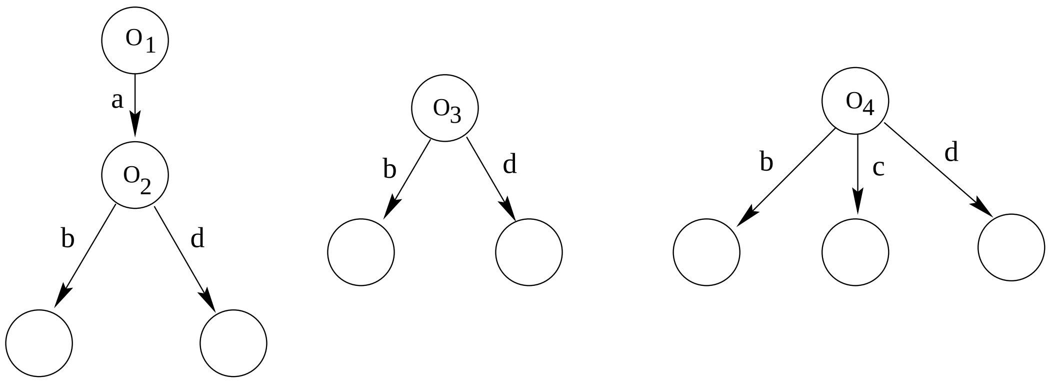

Figure 6.4: Example database

1 adding up to a defect of 2. For τ_2, we have no deficit and excess of 1 because we have to disregard $link(o_4, c, o_0)$. Thus, the defect is 1.

6.3 Method Summary

The goal of this work is to be able to type approximately a large collection of semistructured data efficiently. We are therefore led to making simplifying assumptions and introducing heuristics to be able to process this large collection in an effective way. In this section, we present the technique in rather general terms. The various steps are detailed in the following sections.

Our method for the approximate typing of semistructured data consists of three stages. As we shall see, there are several alternatives to be considered at each of the three stages. In order to decide which choices are most appropriate for a given database, we need some information about the data. It should be stressed that these choices remain primarily empirical and that the general process should entail user feedback and adapting the technique to the particular application domain.

The gist of the first stage is to assign *every* object to a single *home type*. We use the minimum number of home types such that every object fits its home type *perfectly* (with no defect). The process of partitioning objects into a collection of home types is similar in spirit to bisimulation [Mil89]. (However, some of the possible variations for this stage yield collections that differ significantly. For example, we could decide to have 'selected' objects with multiple home types.)

In the second stage, we address the optimization problem of reducing the number of types, and thus having objects that fit their home types with some defect, while incurring the lowest cumulative defect. This stage is the hardest both computationally and conceptually. We show that the general optimization problem is NP-hard even for a simple class of semistructured data corresponding to bipartite graphs. There are, however, techniques and heuristics adapted from *k-clustering* [KPR98, Hoc82] that allow efficient and near-optimal treatment of the problem. We also discuss the sensitivity of the solution with respect to the final number of types.

The third and final stage of our method is about recasting the original data within the chosen types. Ideally, the greatest fixpoint semantics of the typing program (consisting of the chosen types) should be employed. However, some of the techniques described in the second stage do not mix well with the fixpoint semantics. For example, some objects may be assigned to more than one particular home type. Such objects don't have all typed links required by their home types. We present ways of resolving the incompatibilities and discuss some additional variations.

6.4 Stage 1: Minimal perfect typing

In this section, we present an algorithm for deriving a perfect (with no defect) typing program from semistructured data. In this program, every complex object has a type that is based on its local picture. The resulting object partitioning of the minimal perfect typing program is related to the partition obtained through bisimulation. We discuss this relationship towards the end of this section.

6.4.1 Assuming a unique role

In this section, we assume that each object lives in a unique class. We will remove this restriction later.

Given some database D, the *minimal perfect* typing program $\mathcal{P}_D$ is constructed as follows:

1. First construct a program $\mathcal{Q}_D$ as follows. Let $o_1 \dots o_N$ be the complex objects. For each complex objects o_k, assign a unique type predicate $type_k$. The rule for $type_k$ will contain $\overset{\leftarrow i}{\ell}$ iff there is an edge labeled ℓ from o_i to o_k, and $\overset{\rightarrow i}{\ell}$ if there is an edge labeled ℓ from o_k to o_i. The rule for $type_k$ will also contain $\overset{\rightarrow 0}{\ell}$ iff there is an edge labeled ℓ from o_i to some atomic object.

2. Compute the *greatest fixpoint* M of $\mathcal{Q}_D$ for D. Let $\equiv$ be the equivalence relation on the types $\{type_k \mid k \in [1..N]\}$ defined by $type_i \equiv type_j$ if $M(type_i) = M(type_j)$. The types of $\mathcal{P}_D$ will be the equivalence classes of $\equiv$, say $\tau_1 \ldots \tau_n$.

3. The new program $\mathcal{P}_D$ is obtained by choosing for each τ_i, a type $type_k$ in τ_i and replacing, in the rule r for $type_k$, each type $type_j$ by its equivalence class $[type_j]$ according to $\equiv$. The home type of o_k becomes $[type_k]$.

Lemma 6.1 *The following property is useful in finding the equivalence classes of types (Step 2 above):*
$type_i \equiv type_j$ *iff* $o_j \in M(type_i) \wedge o_i \in M(type_j)$

The following example illustrates the algorithm for finding the *minimal perfect* typing program.

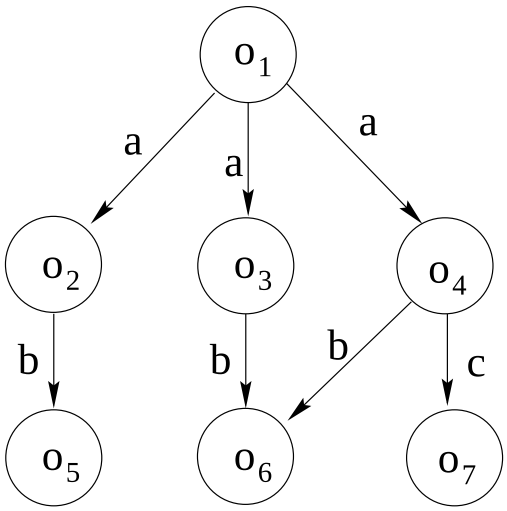

Figure 6.5: Simple semistructured database D.

Example 6.2 *Consider the simple database D in Figure 6.5. This database contains 4 complex and 3 atomic objects. The program $\mathcal{Q}_D$ constructed in (1) of the algorithm is:*

$$
\begin{aligned}
type_1 &= \overrightarrow{a}^{2}, \overrightarrow{a}^{3}, \overrightarrow{a}^{4} \\
type_2 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0} \\
type_3 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0} \\
type_4 &= \overleftarrow{a}^{1}, \overrightarrow{b}^{0}, \overrightarrow{c}^{0}
\end{aligned}
$$

The greatest fixpoint M for $\mathcal{Q}_D$ obtained in (2) is:

$$M(type_1) = \{o_1\}, M(type_2) = M(type_3) = \{o_2, o_3, o_4\}, M(type_4) = \{o_4\}.$$

Let $[type_1] = \tau_1$, $[type_2] = [type_3] = \tau_2$ and $[type_4] = \tau_3$. The program $\mathcal{P}_D$ is:

$$\begin{array}{rcl} \tau_1 & = & \overrightarrow{a}^3, \overrightarrow{a}^2 \\ \tau_2 & = & \overleftarrow{a}^1, \overrightarrow{b}^0, \overrightarrow{c}^0 \\ \tau_3 & = & \overleftarrow{a}^1, \overrightarrow{b}^0 \end{array}$$

The home type object partition for this program is:

- *τ_1 is the home type for o_1.*
- *τ_2 is the home type for o_2 and o_3.*
- *τ_3 is the home type for o_4.*

Note also that for recursive datalog programs the greatest fixpoint semantics is needed to derive the intuitively correct classification. In the above program the least fixpoint will be empty. For nonrecursive datalog programs the greatest and the least fixpoints coincide.

There is a straightforward method for computing the greatest fixpoint for a program $\mathcal{P}$. First, assign every type to every object and call this database M^{all}. Then compute $\mathcal{P}(M^{all} \cup link \cup atomic)$. Keep applying $\mathcal{P}$ to the result of the last application until no change occurs.

The partition of the objects at this stage is a specialization of the partition induced by *bisimulation* [BDHS96, Suc96, Mil89]. Intuitively, two nodes are *bisimilar* if after the (possibly infinite) unfolding from each vertex and after duplicate elimination for subtrees, the two resulting (possibly infinite) regular trees are identical. A subtlety is that we do consider here both incoming and outgoing edges, which leads also to introducing edges corresponding to incoming edges when unfolding a vertex. Bisimulation turns out to be relatively easy to compute. First, we consider that all objects are in a unique class c_0. At some stage, suppose that the objects are separated in a partition $\pi_1, ..., \pi_m$. If for some classes π_i, π_j and some label ℓ, there are objects in π_i that have ℓ edge going to objects in π_j and some that do not, one can split π_i in two. (Similarly, if some objects in π_i have incoming ℓ edges from π_j and some that do not.) This yields a more refined partition. Ultimately, this provides a partition of the set of objects and a type based on this partition.

6.4.2 Multiple roles

As the result of the minimal perfect typing so far, each object has its *home* type based on the object's local picture. Note however that the types defined by the minimal perfect typing program may still overlap. The reason is that the program does not contain negation. Thus, objects that have more typed links than required for a given type will also be assigned to that type, even though it is not their home type. Such assignments are the style of ODMG inheritance but somewhat richer, since our description of the locality of an object includes not only its outgoing edges (as in ODMG) but its incoming edges as well.

In the context of semistructured data, it seems often compulsory to remove the home type assumption that states that, for each object, there is a type that fully describes it. Objects may have multiple roles, and each role may come equipped with a set of possibly overlapping attributes. For example a person may be an employee, a soccer player, a foreigner, a friend, etc., and each of its possible roles may come equipped with a pattern of incoming and outgoing edges. We want to avoid the combinatorial explosion of introducing employee-soccer-player-foreigner, employee-foreigner-friend, etc. Indeed, forcing each object to be in a single type would artificially increase the number of types or the error of the typing.

At this stage, we can identify complex types that can be expressed as a conjunction of several simpler types. By simpler we mean having fewer typed links in their definition. The home objects for the complex types can then be assigned to each of the simpler types that cover the complex one. Thus, at the end of this operation we will have an overlapping collection of types. The following example illustrates the main idea.

Example 6.3 *Consider the database in Figure 6.6. Its natural perfect typing program is:*

$$\begin{aligned}
type_1 &= \overrightarrow{Name}^{0}, \overrightarrow{Country}^{0}, \overrightarrow{Team}^{0} \\
type_2 &= \overrightarrow{Name}^{0}, \overrightarrow{Country}^{0}, \overrightarrow{Team}^{0}, \overrightarrow{Movie}^{0} \\
type_3 &= \overrightarrow{Name}^{0}, \overrightarrow{Country}^{0}, \overrightarrow{Movie}^{0}
\end{aligned}$$

In the greatest fixpoint, $type_1$ contains o_1 and o_2; $type_2$ contains o_2; $type_3$ contains o_2 and o_3. Thus, even if we delete $type_2$ every object will still be assigned to at least one type. In

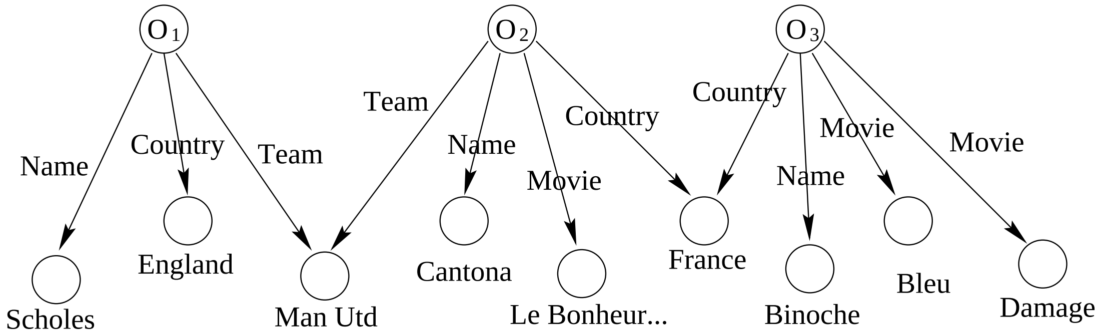

Figure 6.6: Soccer and movie stars.

that case, o_2 will loose its original home type but will be assigned two home types, namely $type_1$ and $type_3$.

It should be observed that although the introduction of new very general types may sometimes be useful, overdoing it may lead to some "atomization" of the information. Intuitively, one would like to avoid describing a person as some object that is in a class *has-name* and in a class *has-address* and in a class *has-spouse*.

However, the decision whether to eliminate some complex types covered by several simpler ones can be deferred to the next stage of our method — the clustering of types. In the clustering stage, types with many typed links and few home-type objects will tend be coalesced with simpler types.

6.5 Stage 2: Clustering

In most cases, the *minimal perfect* typing program will have too many types to be useful as a summary of the data set. There will be many 'similar' types that intuitively can be collapsed into one thus dramatically reducing the size and complexity of the typing program. In this section, we outline how to transform the typing program to reduce the number of types while keeping the defect (excess + deficit) low.

6.5.1 The general problem

The optimization problem that we consider in this stage is similar to k-clustering [Hoc82]. Every home type along with its weight (the number of objects having this type as their home type) is a point on a hypercube. The dimensions of the hypercube are the different typed links found in the minimal program from Stage 1. The general form of the problem, however, is more complex than k-clustering, because deciding to coalesce several types has the effect of projecting all points on the hypercube on several of its diagonals and thereby reducing the dimensions. Consider the following example.

Example 6.4 *Consider the following four rather similar types:*

$$\begin{array}{lllllll} \tau_1 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 3}{b} & \qquad & \tau_2 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 4}{b} \\ \tau_3 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b} & \qquad & \tau_4 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 2}{b} \end{array}$$

Initially, all 4 types are different. However, if we coalesce either τ_1 and τ_2 or τ_3 and τ_4, the remaining two types become identical. Of course, in this case, it doesn't matter which pair is chosen first. However, there are situations where the order of coalescing has a significant effect on the quality of the result.

6.5.2 Distance function between types

There are many ways to define the distance between two types. We argue that while the fine tuning of the parameters of a specific function is very domain specific, the general properties of the distance function are universal.

Consider two types τ_1 and τ_2 and their definitions. The simplest and most natural distance function seems to be the *Manhattan path* between the two type points on the binary hypercube defined by the typed links in their definitions. In simpler terms, the distance is the number of typed links in the symmetric difference between the bodies of their rule definitions. We denote this distance, which is the basis of more complex functions considered later, by $d(\tau_1, \tau_2)$.

Example 6.5 *Consider the following three types:*

$$\begin{array}{lll} \tau_1 & :- & \overset{\rightarrow 0}{a}, \overset{\rightarrow 2}{b} \\ \tau_2 & :- & \overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b} \\ \tau_3 & :- & \overset{\rightarrow 2}{b}, \overset{\leftarrow 1}{b}, \overset{\leftarrow 3}{b}\ . \end{array}$$

For τ_1, τ_2, the symmetric difference consists of $\{\overset{\rightarrow 2}{b}, \overset{\leftarrow 1}{b}\}$, so $d(\tau_1, \tau_2) = 2$. For τ_1, τ_3, the symmetric difference consists of $\{\overset{\rightarrow 0}{a}, \overset{\leftarrow 1}{b}, \overset{\leftarrow 3}{b}\}$, so $d(\tau_1, \tau_3) = 3$. And $d(\tau_2, \tau_3)$ is also 3.

Although this simple distance function appears to be very natural, it does not take into account the weight of the types (the number of objects having the type as their home type). We need to use a more complex *weighted distance* δ that should be a function of the Manhattan distance d, and the weights of the two types w_1 and w_2. The distance δ is not symmetric because $\delta(w_1, w_2, d)$ measures the cost of moving type objects of type τ_2 to τ_1. It seems desirable to have the following properties for such a distance:

increasing in d This property is based on the intuition that it is better to collapse 'similar' types, i.e., such that there are very few typed links in one and not in the other.

decreasing in w_1 This is based on the intuition that the expected noise around some class of object should be proportional to the number of objects in the class. In other words, if the class has a very large extent, we may expect a lot of objects that almost fit in it but not quite and should be willing to correct them.

increasing in w_2 The intuition behind this last property is that large collections of similar objects are likely to form types and thus should not be moved to other types (unless the other type is much bigger and thus the previous property kicks in).

These three properties are clearly related to the overall goal of minimizing the defect.

There are several possible functions that seem reasonable choices even though some of them don't satisfy all three properties listed above:

$$\begin{array}{ll} \delta_1(w_1, w_2, d) = d * w_2 & \delta_2(w_1, w_2, d) = L^d/(w_1 * w_2) \\ \delta_3(w_1, w_2, d) = (w_1 * w_2)^{1/d} & \delta_4(w_1, w_2, d) = (w_2/w_1)^{1/d} \\ \delta_5(w_1, w_2, d) = L^d * w_2 & \end{array}$$

where L is the total number of different typed links in the typing program obtained at the end of Stage 1. Clearly, the choice of a distance function seriously affects the results of the typing. The following example shows that deciding on the particular parameters for a given function is domain specific.

Example 6.6 *Suppose at the end of Stage 1 there are only three types.*

- *100000 objects of type* $\tau_1 = \vec{a}^{\,0} \vec{b}^{\,0}$
- *1000 objects of type* $\tau_2 = \vec{a}^{\,0} \vec{b}^{\,0} \vec{c}^{\,0}$

- *100 objects of type* $\tau_3 = \vec{a}^0 \vec{b}^0 \vec{\ell_1}^0 \ldots \vec{\ell_k}^0$

Suppose that we want to end up with only two types at the end of Stage 2. We implicitly assume that one extra type will be the empty set allowing us to chose not to type some objects by assigning them to the empty set type. For $k = 1$*, the best solution will be to move* τ_3 *to* τ_1*. Similarly, for a big* k*, e.g.,* $k > 15$*, the best solution is to move* τ_2 *to* τ_1*. In between, there is a range for* k *such that the best solution is to move* τ_3 *to the empty set type, i.e, to not classify those 100 objects with a home type* τ_3*. The two cut-off points depend on the distance function that is chosen and are clearly application dependent.*

Note that the distance function δ_1 resembles our definition of defect introduced in Section 6.2. While it measure the defect exactly for a single coalescing, when we have a series of coalescing of types it only provides an upper bound on the defect of the final program.

Clustering algorithm Since finding the optimal k types is NP-hard we have to employ heuristics in order to solve the problem. In our experiments we used a greedy algorithm because of it lower time complexity and implementation ease. Furthermore, under certain assumptions, the greedy algorithm gives an $O(\log n)$-approximation of the best solution [Hoc82].

To conclude this section, we consider a special case that is somewhat easier and an alternative to the clustering in general.

Bipartite graphs An important special case is when all typed links point to atomic objects which happens when the graph is bipartite. Bipartite graphs result from relational data or when the data comes from a file of records. Then each type is defined by the set of labels on the outgoing links, i.e. the attributes in the relational case. The problem is much simpler. However, even in this simple case, one can show that finding the best typing with k types (for some fixed k), where "best" is defined by minimizing the defect, is still NP-hard.

Variation to k-clustering A different approach is first to consider the types after Stage 1 without their weights. Using some measure of the relative importance of an attribute within a set of attributes (e.g. the *jump* function [NAM97]) we can find the best k clusters of the types and only use the weights within a cluster to determine its type definition corresponding to its center. However, this approach may run into problems if there are many outliers and the hypercube is densely populated.

6.6 Stage 3: Recasting

In the third stage we allow objects to be in types other than their home type(s) if they satisfy the appropriate type predicates. At this stage however we do not account for the excess or deficit. Thus, at the end of the third stage we have typed approximately all objects with k types at some defect cost. Note that the first stage is independent of the choice of k. Thus, we can support a sliding scale mechanism where the scale is k and the result is the best k types and the corresponding defect. In fact, our experiments suggest that using this approach yields better results and provides additional insight into the data. We present more detailed discussion of this approach in Section 6.7.

When we allow objects to be assigned to types other than their home type(s) we actually have several options depending on whether we only classify objects based on their actual typed links or the ones suggested by their home type assigned at the end of Stage 2.

The typing rules for objects that have not been used to derive the typing program are rather simple. First we assign a new object to all types that it satisfies completely. If the object cannot be assigned any type precisely, then we assign it to the closest type to it, in terms of the simple distance function d. Of course, if we have many new objects we may wish to reconsider the the current typing program. Deciding how many new objects is too many and recomputing efficiently the typing program are open problems.

6.7 Experimental results

In this section we present our experimental results. While performance (in terms of time) is an important consideration in our work, the main focus of the experiments was the quality of the results. Indeed, understanding when the various options in our algorithm work best and how is a prerequisite for designing efficient data structures and optimization. In this performance study we used extensively synthetic data. We also show some results on a operational data set. Note the using synthetic data is attractive for the purpose of evaluating the quality of the typing in several ways. First, we are able to compare the types produced by our algorithm with the *intended* type in the data specification. Second, we are

FromObj	ToObj	Label
C1	*A1*	a
C1	*A5*	b
C2	*A3*	a
C3	*C1*	c
C3	*A2*	b
C4	*A1*	b
C5	*C2*	c

Figure 6.7: Example of synthetic data.

able to measure the effects of various perturbation of the data on the the typing results.

6.7.1 Generating Synthetic Data

The main idea behind the synthetic data is to use type definition with probability attached to their typed links and then produced random instances according to those probabilities. The following example illustrate the data generation process.

Example 6.7 *Consider the following type specification. There are two types in addition to the standard atomic type. In order to simulate imperfect data we generate objects in the following probabilistic manner. Objects of the first type have a link labeled 'a' to an atomic object with probability 0.9 and a link labeled 'b' to an atomic object with probability 0.5. Objects of the second type have a link labeled 'c' to an object of the first type with probability 0.8 and a link labeled 'b' to an atomic object with probability 0.9. Figure 6.7 gives a small database that might be generated from this type specification.*

The results of running our typing algorithm for several synthetic data sets are captured in Table 6.1. The distance function used in the clustering stage is the weighted Manhattan distance. The clustering is done by a greedy algorithm.

We run experiments on 4 different synthetic datasets (DB Nos. 1,3,5,7). For each dataset we denote whether its corresponding graph is bipartite and whether the intended types are overlapping, i.e, have typed links in common. We also consider a slight perturbation of each dataset (DB Nos. 2,4,6,8) where we delete randomly a few links in the graph and then add some randomly labeled links.

Synthetic Data							*Typing*		
DB No	*Bi ?*	*Overlap ?*	*Perturb ?*	*Intended Types*	*Objects*	*Links*	*Perfect Types*	*Optimal Types*	*Defect*
1	Y	N	N	10	1500	2909	30	10	225
2	Y	N	Y	10	1500	2958	52	10	307
3	Y	Y	N	6	950	2409	19	6	239
4	Y	Y	Y	6	950	2442	35	6	283
5	N	N	N	5	400	726	317	5	181
6	N	N	Y	5	400	749	341	5	310
7	N	Y	N	5	400	775	375	5	291
8	N	Y	Y	5	400	795	381	5	333

Table 6.1: Synthetic data results.

The main observation from the results is that slight perturbation of the dataset results in a dramatic increase of the number of perfect types, while the effect on the optimal approximate typing is relatively small. Another observation is that datasets with bipartite graphs are much easier to handle compared to regular graphs.

6.7.2 Sensitivity Analysis

There is clearly a trade-off between the defect and the simplicity of the typing program. For example, the minimal perfect typing program has no defect but has too many types. On the other side of the scale, if we choose to have only one type the defect will be huge unless we are dealing with very regular data. We conjecture that for non-random semistructured data there is usually an optimal number (or a small range) of types. Figure 6.8 show the defect and the total distance used in our typing method as a function of the number of types in the approximate typing. The distance function is the weighted Manhattan distance between the types. As expected, there is a small range of types (6-10) that yields optimal tradeoff between number of types and defect. The optimal typing with 6 types is shown in Figure 6.1.

The existence of optimal range of number of types suggests that an interactive approach to typing semistructured data will work best. Instead of deciding in advance on a fixed number of types in the approximate typing it is better to explore several different values, ranging from as many as in the minimal perfect typing to perhaps just 1. Note that the

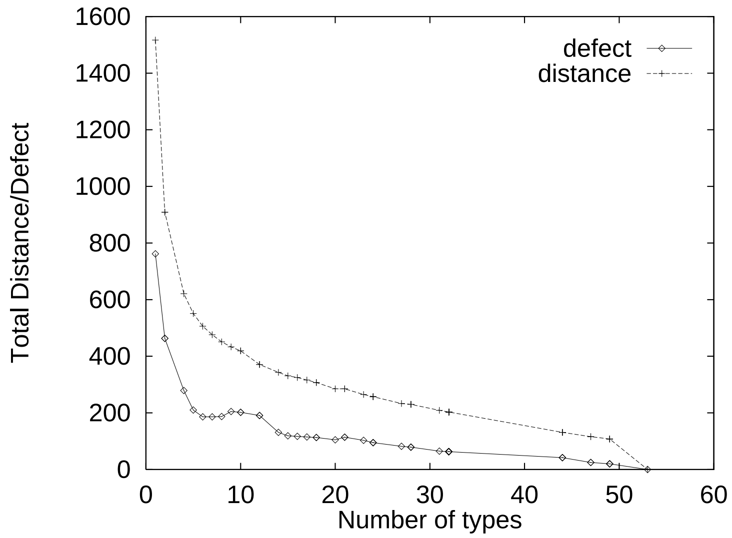

Figure 6.8: Sensitivity graph for DBG data set.

algorithm can be adapted such that the we find sequentially the best fit with k types starting from the number of types in the perfect typing. Thus, the algorithm can find the optimal tradeoff point and suggest a "natural" typing (or a small set). If the results are unsatisfying because of too much defect or too many types, the algorithm can keep reducing the number of types or revert to a typing with more types but less defect. However, we feel that having hard limits on the number of types or the defect, without having knowledge of the data is unreasonable.

6.8 Concluding Remarks

In this chapter, we presented a method for extracting schema from semistructured data. The schema is in the form of a monadic datalog program with each intensional predicate defining a separate type. We asserted that in the context of semistructured data it is imperative to allow for some defect when objects are typed. This assertions was supported by the experimental results on both operational and synthetic data. The perfect typing (with no defect) was shown to be much bigger than the approximate typing produced by our method. Indeed, in some cases the perfect typing was of roughly the same size as the data which precludes its practical use. In contrast, the size of the approximate typing can always be reduced to a desired range. Our experiments suggest that even better results can be obtained by considering the defect as a function of the number of types in approximate typing and choosing an optimal range.

Chapter 7

Conclusions

Data mining — the application of methods to analyze very large volumes of data in order to discover new knowledge — is rapidly finding its way into mainstream computing and becoming commonplace in such environments as finance and retail, in which large volumes of cash register data are routinely analyzed for user buying patterns of goods, shopping habits of individual users, efficiency of marketing strategies for services and other information. In this thesis, we presented data mining techniques that contribute towards a comprehensive solution for both structured and semistructured data.

The contributions of this thesis for data mining of structured data include:

- Framework, called query flocks, for declarative formulation of a large class of data mining queries in a uniform manner.
- Methods, called query flock plans, for systematic optimization and efficient processing of query flocks with monotone filters.
- Architecture for tightly-coupled integration of query flocks and query flock plans with relational DBMS.

The contribution of this thesis for data mining of semistructured data include:

- Definition of the problem of discovering structure from semistructured data.
- Methods for mining precise structural summaries from semistructured data.
- Methods for mining approximate schemas in the form of datalog programs that account for noisy data.

7.1 Future Work

The are several directions for future work. In Chapter 3 we showed that our optimization methods work for query flocks that consist of a single conjunctive query and a monotone filter condition. We plan to expand the techniques to cover other types of filters such as the high-confidence-without-high-support condition [FMU00, MCD+00]. Such expansion will involve adopting some form of hashing, perhaps similar to [FSGM+98], and allowing multiple auxiliary relations for the same set of parameters. We also plan to consider the optimization issues for query flocks that include unions of conjunctive queries.

An orthogonal avenue of future research is to consider cost-based optimization techniques. Such undertaking necessarily involves devising novel methods for collecting and using statistics that can be used to estimate results of aggregations. These new methods must be combined with the existing techniques in standard query optimization and perhaps augmented with dynamic query plans.

In Chapter 6 we mentioned that our methods for structure discovery can benefit from incorporating existing information about the explicit partial structure of semistructured data. In the context of XML, it is an interesting and important problem to consider adapting our algorithms to the particularities of XML and then consider using the information about the explicit schema of the data provided by the DTD to accelerate our algorithms.

Bibliography

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of VLDB*, pages 39–51, Taipei, Taiwan, 1993.

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of ICDT*, pages 1–18, Delphi, Greece, January 1997.

[ABS99] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco, California, 1999.

[AD99] B. Adelberg and M. Denny. Nodose version 2.0. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 559–561, Philadelphia, Pennsylvania, June 1999.

[Ade98] B. Adelberg. Nodose-a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 283–294, Seattle, Washington, June 1998.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading, Massachusetts, 1995.

[AIS93] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 207–216, May 1993.

[Apt91] K.R. Apt. *Logic Programming, Handbook of Theoretical Computer Science*. J. Van Leeuwen, Elsevier, 1991.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. Technical report, Dept. of Computer Science, Stanford University, 1996. Available by anonymous ftp to `db.stanford.edu`.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[BCK+94] G. Blake, M. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. Tompa. Text/relational database management systems: Harmonizing SQL and SGML. In *Proceedings of the First International Conference on Applications of Databases*, pages 267–280, Vadstena, Sweden, 1994.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Addind structure to unstructured data. In *Proceedings of ICDT*, pages 336–350, Delphi, Greece, January 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference*, pages 505–516, Montreal, Canada, June 1996.

[BGG97] E. Borger, E. Graedel, and Y. Gurevich. *The classical decision problem.* Speinger-Verlag, Berlin Heidelberg, 1997.

[BK94] C. Beeri and Y. Kornatski. A logical query language for hypermedia systems. *Information Sciences*, 77, 1994.

[BMTU97] S. Brin, R. Motwani, D. Tsur, and J. Ullman. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 255–264, Tucson, Arizona, June 1997.

[Bun97] P. Buneman. Semistructured data: a tutorial. In *Proceedings of PODS*, Tucson, Arizona, May 1997.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1994.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis project: Integration of heterogeneous information sources. In *Proceedings of 100th Anniversary Meeting of the Information Processing Society of Japan*, pages 7–18, Tokyo, Japan, October 1994.

[CHNW96] D. W. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of ICDE*, pages 106–114, New Orleans, Louisiana, February 1996.

[CM77] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of 9th Annual ACM Symposium on the Theory of Computing*, pages 77–90, Boulder, Colorado, May 1977.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20st International Conference on Very Large Data Bases*, pages 354–366, Santiago, Chile, September 1994.

[CS96] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 167–182, Avignon, France, March 1996.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a Web-site management system. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 26–33, Tucson, Arizona, May 1997.

[FMU00] S. Fujiware, R. Motwani, and J. Ullman. Dynamic miss-counting algorithms: Finding implication and similarity rules with confidence pruning. In *Proceedings of ICDE*, San Diego, California, March 2000.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proceedings of the Twenty-Fourth International Conference on Very-Large Databases*, pages 299–310, New York, New York, 1998.

[GM99] S. Grumbach and G. Mecca. In search of the lost schema. In *Proceedings of the International Conference on Database Theory*, pages 314–331, Jerusalem, Israel, January 1999.

[GMUW00] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, Upper Sadle River, New Jersey, 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to xml: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, June 1999.

[Gra87] G. Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, University of Wisconsin-Madison, 1987.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[GW99] R. Goldman and J. Widom. Approximate dataguides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.

[HKK97] E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 277–288, Tucson, Arizona, June 1997.

[Hoc82] D.S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[Hop71] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of ACM*, 29(3):699–717, 1982.

[KPR98] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, January 1998.

[LS93] A. Y. Levy and Y. Sagiv. Queries independent of update. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 171–181, Dublin, Ireland, August 1993.

[Man97] H. Mannila. Methods and problems in data mining. In *Proceedings of International Conference on Database Theory*, pages 41–55, Delphi, Greece, January 1997.

[MCD+00] R. Motwani, E. Cohen, M. Datar, S. Fujiware, A. Gionis, P. Indyk, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proceedings of ICDE*, San Diego, California, March 2000.

[Mil89] Robin Milner. *Communication and concurrency.* Prentice Hall, 1989.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD International Conference*, pages 295–306, Seattle, Washington, June 1998.

[NLHP98] R. Ng, L. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 13–24, Seattle, Washington, June 1998.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of ICDE*, pages 79–90, Birmingham, U.K., April 1997.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of ICDE*, pages 251–260, Taipei, Taiwan, March 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Singapore, December 1995.

[SA95] R. Srikant and R. Agrawal. Fast algorithms for mining association rules. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 407–419, Zurich, Switzerland, September 1995.

[STA98] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 343–354, Seattle, Washington, June 1998.

[Suc96] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proceedings of VLDB*, pages 227–238, 1996.

[TUC+98] S. Tsur, J. Ullman, C. Clifton, S. Abiteboul, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1–12, Seattle, Washington, June 1998.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I - Fundamental Concepts*. Computer Science Press, Rockville, Maryland, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II - The New Technologies*. Computer Science Press, Rockville, Maryland, 1989.

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1997.

[WL97] K. Wang and H. Liu. Schema discovery for semistructured data. In *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pages 271–274, Newport Beach, California, August 1997.

[WL00] K. Wang and H. Liu. Discovering structural association of semistructured data. *IEEE Transactions on Knowledge and Data Engineering*, 2000. To appear.

[XML98] Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[YL95] W. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 345–357, Zurich, Switzerland, September 1995.

[ZO93] X. Zhang and M. Z. Ozsoyoglu. On efficient reasoning with implication constraints. In *Proceedings of International Conference on Deductive and Object-Oriented Databases*, pages 236–252, Phoenix, Arizona, December 1993.