Querying Semistructured Data based on Schema Matching

André Bergholz*†
Humboldt-University Berlin
bergholz@dbis.informatik.hu-berlin.de

Johann Christoph Freytag
Humboldt-University Berlin
freytag@dbis.informatik.hu-berlin.de

Abstract

Traditional database management requires design and ensures declarativity. In the context of semistructured data a more flexible approach is appropriate due to missing schema information. In this paper we present a query language based on schema matching. Intuitively, a query is a pair consisting of what we want and how we want it. We propose that the former can be achieved by matching a (partial) schema and the latter by specifying additional operations. We describe in some detail our notion of schema which covers various concepts such as predicates, variables and paths. We outline the optimization potential that this modular approach offers and discuss how we use constraints for query processing.

1 Introduction

Traditional database management requires design and ensures declarativity. Semistructured data, "data that is neither raw data nor strictly typed", lacks a fixed and rigid schema [Abi97]. Often their structure is irregular and implicit. Examples for semistructured data include HTML files, BibTEX files or genome data stored in ASCII files. Recently, XML has emerged as a common syntactical representation for semistructured data. To us, the key problem of semistructured data is to move from *content-based querying*, i.e. the UNIX `grep`-command or simple WWW search engines, to *structure-based querying*, i.e. querying in an SQL-like manner.

*Contact author: André Bergholz, LFE Datenbanken und Informationssysteme, Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany (phone: +49-30 2093 3024)

†This research was supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316)

Before we present the idea of the paper we take a look at relational systems. We identify three layers: the operational layer, the schema layer and the instance layer. The tuples form the instance layer and the tables form the schema layer. On the operational layer we find queries, views, constraints etc. We notice that the items of the operational layer are expressed using the items of the schema layer, i.e. queries are expressed using the tables. We would like to adapt this framework of querying for semistructured data. As we have learned, the serious problem of semistructured data is its lack of complete, known-in-advance structure. We observe that in this query framework for relational data every item on the instance layer (i.e. every tuple) belongs to exactly one item on the schema layer (i.e. to exactly one table). This constraint certainly has to be relaxed in the context of semistructured data.

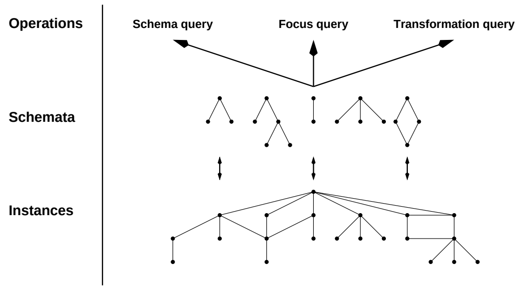

Figure 1: A new query framework for semistructured data

Figure 1 shows some graph schemata in the middle layer that conform to some parts of the database at the bottom layer. There is no further restriction, i.e. a schema can have an arbitrary number of instances and instances can conform to an arbitrary number of schemata. The operational layer is based on this conformity between schema and object.

How can we find those schemata in the middle layer? The simplest way is to obtain them from a database designer. While this is not in the spirit of semistructured data this kind of data is – for good reasons – called semistructured rather than unstructured. At least some parts of the data can potentially be modeled. On the other hand, a query uses both the schema and the operational layer. This leads to the analogy of seeing a query consisting of a "What"-part (i.e. a schema) and a "How"-part (i.e. an operation). In the relational world we can consider a selection to correspond to a "What" and a projection correspond to a "How". Now an obvious approach is to cache the "What" 's, i.e. to extract schemata out of queries. Those schemata are useful for two main purposes. First, they can give users a clue about the content of a database. Second, they can be used for query optimization as we will show later on. Note, that schemata that are good for the former are not necessarily good for the latter and vice versa.

The first and foremost advantage of this approach is that a database system designed in this way reflects the degree of structure of the database on many levels. If the database is rather well structured there will be schemata with many instances. This will give users a pretty good idea about the data, and the performance of the system will be good as well. If, however, the database is not that well structured there will be only some useful schemata. Thus, the user will not get full knowledge about the database and the performance will suffer as well. A related advantage is that the schema layer can serve as an indication of the degree of structure of the database. The following three paradigms shall guide our approach:

1. Answering a query *works without* schema information.
2. Answering a query *benefits from* schema information.
3. Answering a query *induces new* schema information.

This paper is organized as follows. Section 2 presents the syntactical data representation we are using. In Section 3 we define our semantically rich notion of schema that forms the base for the queries that are introduced in Section 4. The second part of the paper deals with query processing based on constraints. This is outlined in Section 5. We conclude with related work and a discussion.

2 Labeled graphs as data representation

In this section we describe the underlying data model of our proposal. It is a graph-based approach because graph models seem to be "the unifying idea in semi-structured data" [Bun97]. We try to be very general and do not require any specific restrictions to our graphs.

Definition 1 (Total directed graph). A tuple $G = (V, A, s, t)$ is a *total directed graph* if V is a set of *vertices*, A a set of *arcs* and furthermore s and t are total functions from A to V assigning each arc its *source* and *target* vertex, respectively.

We also use the term *node* instead of vertex. However, we use the term arc instead of edge to emphasize that we consider directed graphs. In our model two nodes can be linked by more than one arc. Cycles are allowed. The following definition introduces labels on vertices and arcs.

Definition 2 (Labeled directed graph). Let $\mathcal{L}$ be an arbitrary set of *labels*. A tuple $G = (V, A, s, t, l)$ is a *($\mathcal{L}$-)labeled directed graph* if (V, A, s, t) is a total directed graph and $l : V \cup A \longrightarrow \mathcal{L}$ is a total *label function* assigning each vertex and arc a label from $\mathcal{L}$.

An *object* is now a labeled directed graph. We also use the term *database* instead of object. We do this when we talk about a "large" object that is to be queried. Note that we usually denote objects with lower-case letters (i.e. $o_1, o_2, \dots$) but graphs with upper-case letters (i.e. $G_1, G_2, H, \dots$) in order to be consistent with both worlds.

Figure 2 presents an example that we shall use throughout the paper. It shows a semistructured database on persons having names, surnames, a year of birth, a profession etc. Additionally, a sibling relationship relates different people.

For specifying answers to queries we will need the notion of a subobject of a database. This assumes some basic knowledge of partial orders, for an introduction see e.g. [Tro92].

Definition 3 (Subobject). An object $o_2 = (V^{(o_2)}, A^{(o_2)}, s^{(o_2)}, t^{(o_2)}, l^{(o_2)})$ is a *subobject* of $o_1 = (V^{(o_1)}, A^{(o_1)}, s^{(o_1)}, t^{(o_1)}, l^{(o_1)})$ if $V^{(o_2)} \subseteq V^{(o_1)}$, $E^{(o_2)} \subseteq E^{(o_1)}$, $s^{(o_2)} = s^{(o_1)}|_{A^{(o_2)}}$, $t^{(o_2)} = t^{(o_1)}|_{A^{(o_2)}}$ and $l^{(o_2)} = l^{(o_1)}|_{V^{(o_2)} \cup A^{(o_2)}}$. We denote this by $o_2 \subseteq o_1$.

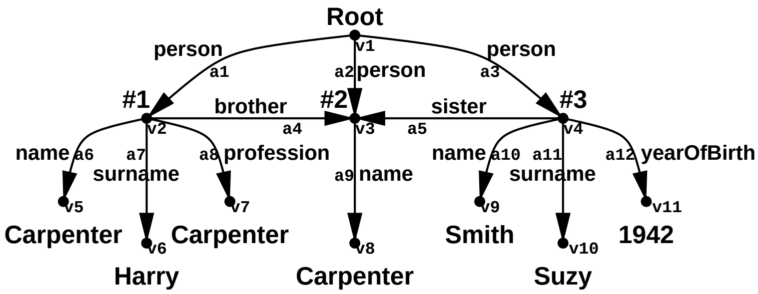

Figure 2: A labeled directed graph

For a given object o we denote the set of all its subobjects by $\mathfrak{P}(o)$.

Lemma 1. *For a given object o the structure $[\mathfrak{P}(o), \subseteq]$ is a partially ordered set, i.e. $\subseteq$ is a reflexive, antisymmetric and transitive binary relation over $\mathfrak{P}(o)$.*

Lemma 2. *For a given object o the structure $[\mathfrak{P}(o), \subseteq]$ is a lattice, i.e. every nonempty subset of $\mathfrak{P}(o)$ has a least upper and a greatest lower bound.*

As an example we show in Figure 3 the Hasse-diagram of all subgraphs of a directed tree with three nodes.

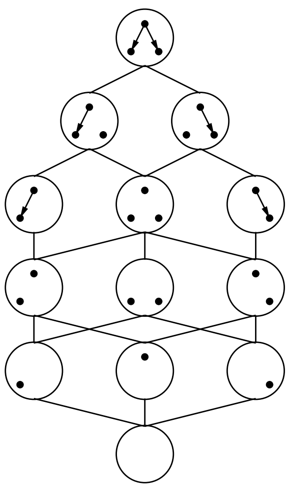

Figure 3: The lattice of all subgraphs of a simple tree

3 Schemata

This section introduces the notions of schema and conformity between schema and object. Informally, a schema is an object that describes a set of objects. In the simpler syntactic framework of the label world this schema concept certainly exists as well. One label might describe a set of other labels. This is frequently done – data types, predicates and regular expressions are examples.

As as first step towards schemata in the graph world we assign schemata from the label world to the elements of the graph. We choose predicates to be the label world schemata.

Definition 4 (Predicate schema). Given a set of unary predicates $\mathcal{P}$, a *predicate schema* (over $\mathcal{P}$) is an object $s = (V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$ where the elements are labeled with predicates ($l : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{P}$).

We give an example in Figure 4. Note that we treat a quoted constant c as an abbreviation for the predicate $X = c$.

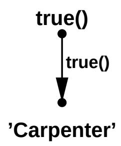

Figure 4: A simple predicate schema

In order to establish a relationship between a schema and the objects described by it we must establish the notion of *conformity* between schemata and objects. Depending on the direction of the mapping we say that we *match* a schema into an object or we *interpret* an object into a schema.

Definition 5 (Naive conformity). A *match* of a predicate schema s into an object o is an isomorphic embedding m of s into o such that for all $x \in V^{(s)} \cup A^{(s)}$ the predicate $l^{(s)}(x)$ holds for $l^{(o)}(m(x))$.

If there exists a match of the schema s into the object o we say that o *conforms* to s and we call o an *instance* (or also a *match*) of s.

Let o be a database, s be a schema and $o_1 \subseteq o$ a match of s. Then every object o_2 with $o_1 \subseteq o_2 \subseteq o$ is also a match of s. Let $\mathfrak{M}^{(s)}(o)$ denote the set of all matches of s in o. Since $\mathfrak{M}^{(s)}(o) \subseteq \mathfrak{P}(o)$ (in set semantics) $[\mathfrak{M}^{(s)}(o), \subseteq]$ is also a partially ordered set. We call a minimal element of this partially ordered set a *minimal match* (or a *minimal instance*) of s in o. We denote the set of minimal matches of s in o with $\mathfrak{M}^{(s)}_{min}(o)$. In Figure 5 we show the same schema as in Figure 4 but together with its minimal matches in the database from Figure 2.

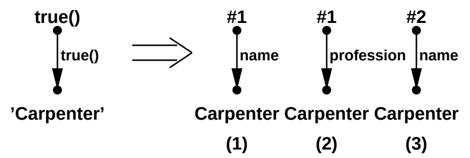

Figure 5: The predicate schema and its minimal matches

We add variables to our schemata in the following manner: Let s be a schema, $\mathcal{V}$ be a set of variables and $v : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{V}$ be a partial mapping from the nodes and arcs in the schema into the variables. For a mapping m to be a match of s into an object o we additionally require for all $x_1, x_2 \in V^{(s)} \cup A^{(s)}$ that if $v(x_1)$ and $v(x_2)$ exist and $v(x_1) = v(x_2)$ then $l^{(o)}(m(x_1)) = l^{(o)}(m(x_2))$, i.e. the labels of the corresponding elements in the match are the same. A predicate schema with variables together with its minimal match is shown in Figure 6. Due to the very nature of semistructured data variables can be used to link data and structural parts of the database.

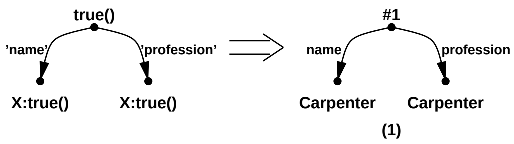

Figure 6: Adding variables

For adding paths to our notion of schema we have to take into account structural aspects of graphs. Let $G = (V, A, s, t)$ be a total directed graph. A *trail* is an arc sequence $(a_{i_1}, \ldots, a_{i_m})$ where all a_{i_j} are distinct and there exist nodes $v_{i_0}, \ldots, v_{i_m}$ such that for all a_{i_j} $s(a_{i_j}) = v_{i_{j-1}}$ and $t(a_{i_j}) = v_{i_j}$ hold. Note that we do not require the $v_{i_0}, \ldots, v_{i_m}$ to be distinct. Thus, the notion of trail is more general than that of a path, yet the number of trails in an arbitrary graph is always finite, which makes it possible to handle cyclic structures. The number of arcs in a trail is called the *length* of the trail. Despite the fact that we are talking about trails we denote the set of all trails in a graph by P and the set of nonempty trails by P^+ because from the intuition point of view we are talking about paths. For a nonempty trail $p_i = (a_{i_1}, \ldots, a_{i_m}) \in P^+$ we introduce a source and target function $s_P, t_P : P^+ \longrightarrow V$ which are defined in a canonical manner as $s_P(p_i) = s(a_{i_1})$ and $t_P(p_i) = t(a_{i_m})$, respectively.

Definition 6 (Corresponding trail graph). The *corresponding trail graph* to a graph $G = (V^{(G)}, A^{(G)}, s^{(G)}, t^{(G)})$ is defined as $G_P = (V^{(G)}, P^{+(G)}, s_P^{(G)}, t_P^{(G)})$.

Intuitively, in the corresponding trail graph the trails are materialized as arcs. This notion is related to the notion of transitive closure of a graph. The only difference between the two notions is that in the transitive closure only one arc is included for every pair of reachable nodes whereas we include an arc for every trail via which they are reachable. Figure 7 shows three examples of directed graphs and their corresponding trail graphs.

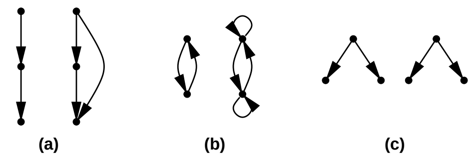

Figure 7: Three directed graphs and their corresponding trail graphs

Lemma 3. *A directed graph is always a subgraph of its corresponding trail graph.*

The lemma holds because there is a natural embedding $a_i \longrightarrow (a_i)$ of the arcs in A into the trails in P^+.

Now we can extend our notion of schema. We introduce two additional functions q_{min} and q_{max} that let us specify length constraints on paths in the matching objects. Furthermore, in order to incorporate the previously mentioned variables, we need a set of variables $\mathcal{V}$ and a variable mapping v.

Definition 7 (Schema). Given a set of labels $\mathcal{L}$ and a set of variables $\mathcal{V}$ a *schema* s is a tuple $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)}, v^{(s)}, q^{(s)}_{min}, q^{(s)}_{max})$ where

1. $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)}, l^{(s)})$ are defined as before,

2. $v : V^{(s)} \cup A^{(s)} \longrightarrow \mathcal{V}$ is the *variable mapping*, a partial mapping from the nodes and arcs in the schema into the variables, and

3. $q^{(s)}_{min} : A^{(s)} \longrightarrow \mathbb{N}^{+}$ and $q^{(s)}_{max} : A^{(s)} \longrightarrow \mathbb{N}^{+} \cup \{+\infty\}$ are *length restrictions.*

Furthermore, if for an arbitrary arc $a_i \in A^{(s)}$ a variable binding $v^{(s)}(a_i)$ exists, then $q^{(s)}_{min}(a_i) = q^{(s)}_{max}(a_i) = 1$ holds.

Of course we have to redefine the notion of conformity between schema and object.

Definition 8 (Conformity). A *match* of a schema s into an object o is an isomorphic embedding of s into o_P, i.e. an isomorphic embedding of $(V^{(s)}, A^{(s)}, s^{(s)}, t^{(s)})$ into $(V^{(o)}, P^{+(o)}, s^{(o)}_P, t^{(o)}_P)$, so that the following properties hold:

1. For all nodes $x \in V^{(s)}$ the predicate $l^{(s)}(x)$ is true for $l^{(o)}(m(x))$.

2. For all arcs $x \in A^{(s)}$ the predicate $l^{(s)}(x)$ is true for the labels $l^{(o)}(y_j)$ of all the arcs y_j in the trail $m(x)$.

3. For all elements $x_1, x_2 \in V^{(s)} \cup A^{(s)}$ for which $v^{(s)}(x_1)$ and $v^{(s)}(x_2)$ exist and $v^{(s)}(x_1) = v^{(s)}(x_2)$, the labels are the same $l^{(o)}(m(x_1)) = l^{(o)}(m(x_2))$.

4. For all arcs $x \in A^{(s)}$ the length of the trail $m(x)$ is at least $q^{(s)}_{min}(x)$ and no greater than $q^{(s)}_{max}(x)$.

If a match between a schema s and an object o exists we say that o *conforms* to s.

If an object o conforms to a schema s we again call o an *instance* (or also a *match*) of s. In order to distinguish between functions that are matches and objects that are matches we also call the functions *match functions.*

The following theorem states that we indeed enhanced our initial notion of schema, i.e. our new notion of schema does not contradict the initial one. Due to space limitations we omit the prove of this theorem.

Theorem 4. *A predicate schema s conforms to an object o in the naive manner if and only if it conforms to o, assuming that $v^{(s)}$ is the empty mapping and $q^{(s)}_{min}$ and $q^{(s)}_{min}$ equal one for all arcs in s.*

Consider the example in Figure 8. There is a '+'-sign on the first arc in the schema. This indicates that the length of the paths (aka trails) it matches is bound by 1 and $+\infty$. So the schema matches everything that emanates from the root and leads to a 'name'-arc.

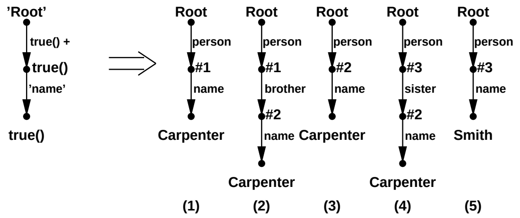

Figure 8: Adding paths

There are some subtleties here. A match of s in o is supposed to be a subobject of o. However, the scope $m(s)$ of the match function m is a subobject of o_P. These subtleties become a serious problem when we want to adapt the definition of minimal match. The notion of minimal match is particularly important for the definition of queries as will be seen in the next section. Consider Figure 9. (We omitted the node labels there because they are not relevant to this problem.)

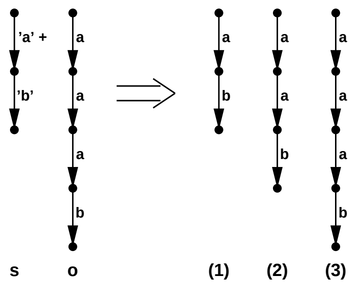

Figure 9: A problem with the minimal matches of paths

The schema on the left is matched to the database next to it. All the three matches are potentially "interesting", but only the first one is minimal since it is a subobject of the other two. Besides, if one of the matches was more interesting than the others, wouldn't it be the one with the longest path, i.e. the one on the right? But we observe that all the three matches result from different match functions. The scopes of their respective match functions are incomparable subobjects of o_P. Thus, we define minimal matches with respect to the match function. In order to achieve this we need a $flatten$-function that takes a subobject of o_P and produces a subobject of o. Informally, $flatten$ decomposes the trails into arcs and adds all their source and target nodes to the node set. Then we can define the set of *minimal matches* of s in o denoted by $\mathfrak{M}^{(s)}_{min}(o)$ as $\{flatten(m(s)) | m \text{ is a match of } s \text{ into } o\}$. We observe that every $flatten(m(s))$ is indeed a match of s in o since s can be embedded into $flatten(m(s))_P$ using m. Furthermore, for every match of s in o (i.e. every element of $\mathfrak{M}^{(s)}(o)$) there is a minimal match in $\mathfrak{M}^{(s)}_{min}(o)$ that is a subobject of the first one. With this revised definition all the three matches on the right hand side of Figure 9 are minimal.

4 Queries

In this section we use the previously introduced schemata to define queries. All queries are based on matching schemata. Whereas the previous section dealt with the "What"-part of a query, this section deals with the "How"-part.

A schema in itself already forms the most simple kind of query. It queries all subobjects of a database that conform to it. However, in such a case we would be interested only in the minimal matches.

Definition 9 (Schema query). A *schema query* is a tuple $q = (s)$ where s is a schema. The *answer* to q with respect to a database o is the set of minimal matches of s in o $\mathfrak{M}^{(s)}_{min}(o)$.

As an example you can imagine any of the schemata from the previous section. With a schema we can formulate conditions that any match must fulfill. This roughly corresponds to a selection in the relational world. However, we would like to have a concept that is comparable to a projection.

Definition 10 (Focus query). A *focus query* is a tuple $q = (s_1, s_2)$ where s_1 is a schema and s_2, the focus, is a subobject of s_1. The *answer* to q with respect to a database o is the union of the minimal matches of s_2 over all minimal matches of s_1 in o, i.e. $\bigcup_{x \in \mathfrak{M}^{(s_1)}_{min}(o)} \mathfrak{M}^{(s_2)}_{min}(x)$.

The example in Figure 10 queries for the surnames of all persons with the name 'Carpenter'. The subschema s_2 is indicated by circles around the elements (in this case just one node) of the superschema s_1.

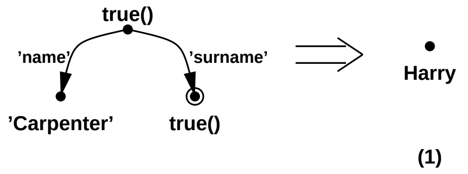

Figure 10: A focus query

However, sometimes we want to restructure the answer completely. Therefore we introduce the *transformation query* where we can specify a graph structure and compute new labels by using terms over the old ones.

Definition 11 (Transformation query). A *transformation query* is a tuple $q = (s, t)$ where s is a schema and t is an object labeled with terms over the elements in s. The answer to q is built by creating for every match of s in o a new object isomorphic to t, labeled with the evaluated terms of t instantiating the terms by using the match.

The example in Figure 11 queries for the age of Suzy Smith. The age is computed from the year of birth.

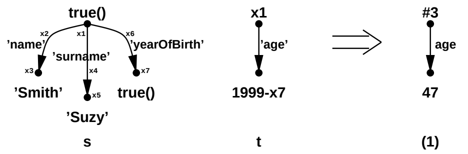

Figure 11: A transformation query

Please note, that schema and focus queries can be expressed as transformation queries.

An obvious limitation of our approach is that we always get one answer per schema match. Thus, we currently do not support aggregation. On the other hand, we think that our approach can easily be extended to cover restructurings of a database.

5 Query processing using constraints

In this section we outline our query processing technique. We focus on finding the matches for a given schema, since this is the computationally challenging part. We start with the description of how to match schemata without any additional schema information given. A great benefit of our approach is that we can use previously matched schemata to speed up query processing. We outline this advantage at the end of the section.

We base our query processing on constraint satisfaction techniques. There are at least two good reasons for this approach. First, the area of constraint satisfaction is well-studied with many techniques and heuristics available. Second, constraint satisfaction problems form a reasonably general class of search problems. Thus, we use a well-established framework for specifying our needs, for adapting our algorithms for richer kinds of schemata and, most importantly, for formulating our query processing based on previously matched schemata.

Constraint satisfaction [Bar98] deals with solving problems by stating properties or constraints that any solution must fulfill. A *Constraint Satisfaction Problem (CSP)* is a tuple (X, D, C) where

- X is a set of *variables* $\{x_1, \ldots, x_m\}$,
- D is a set of finite *domains* D_i for each variable $x_i \in X$ and
- C is a set of *constraints* $\{C_{S_1}, \ldots, C_{S_n}\}$ restricting the values that the variables can simultaneously take. Here the $S_i = (x_{S_{i_1}}, \ldots, x_{S_{i_k}})$ are arbitrary tuples of variables from X and each C_{S_i} is a relation over the crossproduct of the domains of these variables ($C_{S_i} \subseteq D_{S_{i_1}} \times \cdots \times D_{S_{i_k}}$).

Solving a CSP is finding assignments of values from the respective domains to the variables so that all constraints are satisfied. In our context we are interested in finding all solutions of a CSP.

The basic idea is as follows and is summarized in Figure 12. The database graph is transformed into suitable domains and variables are introduced for the elements in the schema. Furthermore, constraints representing the match semantics are introduced. They can be categorized into the ones that represent the label part and the ones that represent the structural part of the match semantics.

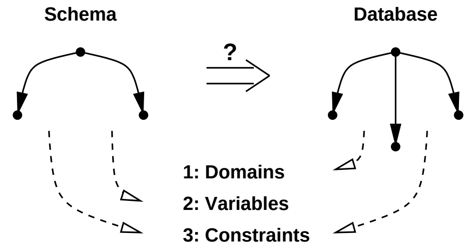

Figure 12: Schema matching by Constraint Satisfaction

We depict the domains of the vertices and arcs from the database graph in Figure 2.

$$D_V = \{v_1, v_2, v_3, \ldots, v_{11}\}$$
$$D_A = \{a_1, a_2, a_3, \ldots, a_{12}\}$$

The example schema in Figure 13 (the same as in Figure 4) gives us the variables and the domain assignments.

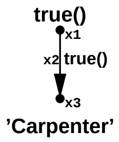

Figure 13: A simple predicate schema

$$X = \{x_1, x_2, x_3\}$$
$$D_1 = D_3 = D_V$$
$$D_2 = D_A$$

Constraints are derived from the labels in the schema ...

$$C^{lab}_{(x_1)} = \{(v_1), (v_2), (v_3), \ldots, (v_{11})\}$$
$$C^{lab}_{(x_2)} = \{(a_1), (a_2), (a_3), \ldots, (a_{12})\}$$
$$C^{lab}_{(x_3)} = \{(v_5), (v_7), (v_8)\}$$

... and the structure of the schema.

$$\begin{aligned} C^{src}_{(x_2,x_1)} = \{&(a_1, v_1), (a_2, v_1), (a_3, v_1), (a_4, v_2), \\ &(a_5, v_4), (a_6, v_2), (a_7, v_2), (a_8, v_2), \\ &(a_9, v_3), (a_{10}, v_4), (a_{11}, v_4), (a_{12}, v_4)\} \\ C^{tar}_{(x_2,x_3)} = \{&(a_1, v_2), (a_2, v_3), (a_3, v_4), (a_4, v_3), \\ &(a_5, v_3), (a_6, v_5), (a_7, v_6), (a_8, v_7), \\ &(a_9, v_8), (a_{10}, v_9), (a_{11}, v_{10}), (a_{12}, v_{11})\} \end{aligned}$$

Our sample CSP has the solutions (v_2, a_6, v_5), (v_2, a_8, v_7) and (v_3, a_9, v_8) for the variables (x_1, x_2, x_3). They correspond to the matches of the schema as shown in Figure 5. Please note that if injectivity of the match is to be ensured, additional constraints must be introduced.

More details about this part of the work (e.g. variables and paths) and about techniques for solving CSPs can be found in [BF99].

We conclude this section by discussing how to find the matches of a schema using previously matched schemata. The basic underlying notion for this approach is the notion of *schema containment*. It is related to the traditional notion of query containment.

Definition 12 (Schema containment). A schema s_1 contains a schema s_2 if for all databases o all matches of s_2 are also matches of s_1.

If s_1 contains s_2

1. All matches of s_2 are also matches of s_1. If we want to find the matches of s_1 and already have the ones for s_2 we can present the *first few matches immediately*. There may exist more matches for s_1, though.

2. Matches of s_2 can only be found among the matches of s_1. If we want to find the matches of s_2 and already have the ones for s_1 we can *reduce the search space.*

Figure 14 shows three schemata. They contain one another left to right.

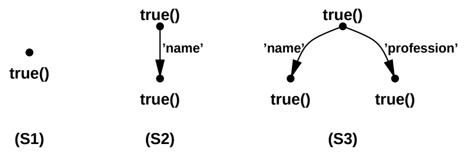

Figure 14: Schema containment

Let us assume that we also define the notion of containment for predicates. p_1 contains p_2 if for all labels x the implication $p_2(x) \longrightarrow p_1(x)$ holds. Now, informally, a schema s_1 contains another schema s_2 if s_1 is a subgraph of s_2 and the predicates of s_1 contain the respective predicates of s_2 and the paths in s_1 are no longer than the respective ones in s_2. The other direction of this implication does not hold.

We reduce the testing of these sufficient conditions for schema containment again to the Constraint Satisfaction Problem. Once we find a schema s' that contains our current schema s we can reduce the search space for the problem of finding the matches of s.

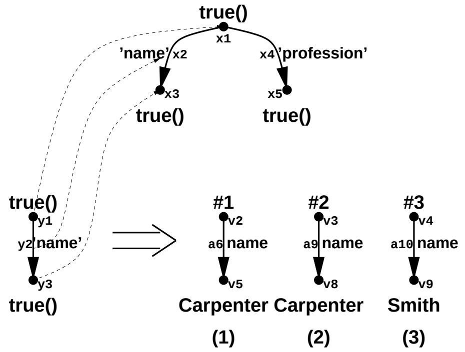

Figure 15: Reducing the search space

In Figure 15 the schema s that is to be matched (the schema on top) is contained in the schema s' on the left. The variables x_1, x_2 and x_3, that are introduced when constructing the CSP for s, correspond to y_1, y_2 and y_3, respectively. In the matches for s' y_1 is matched to v_2, v_3 and v_4, y_2 is matched to a_6, a_9 and a_{10} and y_3 is matched to v_5, v_8 and v_9. Thus, we can construct the reduced domains for x_1, x_2 and x_3.

$$D_1 = \{v_2, v_3, v_4\}$$
$$D_2 = \{a_6, a_9, a_{10}\}$$
$$D_3 = \{v_5, v_8, v_9\}$$

In order to fully capture the containment information we introduce an additional constraint.

$$C^{sol}_{(x_1,x_2,x_3)} = \{(v_2, a_6, v_5), (v_3, a_9, v_8), (v_4, a_{10}, v_9)\}$$

6 Related work

Core issues and basic characterizations of semistructured data are discussed in [Abi97] and [Bun97]. Query languages include Lorel [MAG+97] and UnQL [BDHS96]. XML-QL is similar to our approach in that it uses so called element patterns as the "What"-part of a query [DFF+99]. Work on schema information for semistructured data concentrates on computing an ad hoc complete schema. One example are DataGuides [GW97, GW99]. Another notion of schema is introduced in [BDFS97]. Much related work arises also in the context of query languages suited for the Web and Web-site management systems ([FFK+99], [AMM97], [KS95], [LSS96], [MMM96]). A fundamental work on data stored in files is [ACM93]. Structured files are transformed to databases such that file querying and manipulating by using database technology becomes possible.

We were inspired in our query processing by the area of graph transformations. Graph transformations address the dynamic aspects of graphs. Systems are typically rule-based and can be used to model behavior. Thus, these systems must incorporate techniques for graph pattern matching. Rudolf uses constraint satisfaction techniques for simple graph pattern matching [Rud98]. A more database-like approach to this problem can be found in [Zue93].

[Bar98] and [Kum92] provide an introduction to the field of constraint satisfaction. They give various algorithms, heuristics and useful background information to efficiently solve CSPs. A theoretical study of backtracking algorithms can be found in [KvB97].

7 Conclusion

In this paper we have presented a flexible approach to querying semistructured data. It is based on the intuition that a query consists of a "What"-part and a "How"-part. In contrast to more ad hoc query languages this split allows us to reuse the "what"-part for optimization. We have proposed a general graph model as the underlying data representation. The matching of a (partial) schema (representing the "What") forms the base for querying. We proposed a rather rich kind of schema covering predicates, variables and paths. The "How"-part comes in by defining how to manipulate schema matches. We have outlined how to process a query based on posing constraints. In particular, it is possible to make use of previously matched schemata.

We have started to implement our ideas into a Prolog-based system and are currently switching to the commercial constraint solver ECLiPSe [Ecl]. Additional future work lies in assessing schemata in order to determine "good" schemata for optimization.

Acknowledgement

The authors wish to thank all members of the Berlin-Brandenburg Graduate School in Distributed Information Systems for providing an inspiring environment and for many interesting discussions. We are especially grateful to Felix Naumann for numerous useful comments and for reviewing our manuscript.

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying the file. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1993.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1997.

[Bar98] R. Bartak. Guide to constraint programming, 1998. `http://kti.ms.mff.cuni.cz/~bartak/constraints/`.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, 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 on Management of Data*, pages 505–516, 1996.

[BF99] A. Bergholz and J. C. Freytag. Matching schemata by utilizing constraint satisfaction techniques. In *Proceedings of the Workshop on Query Processing* *for Semistructured Data and Non-Standard Data Formats (in conjunction with ICDT'99)*, 1999.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1997.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the International World Wide Web Conference*, 1999.

[Ecl] ECLiPSe - The ECRC Constraint Logic Parallel System, `http://www.ecrc.de/eclipse/`.

[FFK+99] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 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 (in conjunction with ICDT'99)*, 1999.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World-Wide Web. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 54–65, 1995.

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[KvB97] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A declarative language for querying and restructuring the web. In *Proceedings of the International Workshop on Research Issues in Data Engineering (RIDE)*, 1996.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[MMM96] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World-Wide Web. In *Proceedings of the Conference on Parallel and Distributed Information Systems (PDIS)*, 1996.

[Rud98] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In *Proceedings of the International Workshop on Theory an Application of Graph Transformations (TAGT)*, 1998.

[Tro92] W. T. Trotter. *Combinatorics and Partially Ordered Sets*. The John Hopkins University Press, 1992.

[Zue93] A. Zuendorf. A heuristic for the subgraph isomorphism problem in executing PROGRES. Technical Report AIB 93-5, RWTH Aachen, 1993.