Proc. of the 7th International Conference on Database Theory, 1999.

In Search of the Lost Schema *

Stéphane Grumbach[1] and Giansalvatore Mecca[2]

[1] IASI and INRIA
Rocquencourt BP 105
78153 Le Chesnay, France
stephane.grumbach@inria.fr

[2] DIFA – Università della Basilicata
via della Tecnica, 3
85100 Potenza, Italy
mecca@dia.uniroma3.it

Abstract. We study the problem of rediscovering the schema of nested relations that have been encoded as strings for storage purposes. We consider various classes of encoding functions, and consider the mark-up encodings, which allow to find the schema without knowledge of the encoding function, under reasonable assumptions on the input data. Depending upon the encoding of empty sets, we propose two polynomial on-line algorithms (with different buffer size) solving the schema finding problem. We also prove that with a high probability, both algorithms find the schema after examining a fixed number of tuples, thus leading in practice to a linear time behavior with respect to the database size for wrapping the data. Finally, we show that the proposed techniques are well-suited for practical applications, such as structuring and wrapping HTML pages and Web sites.

1 Introduction

Databases traditionally provide consolidated technologies for organizing and manipulating structured data. However, in recent years, alternative approaches to data organization have imposed themselves as *de facto* standards. A prominent driving force behind this has been the World Wide Web, which delivers information in less structured form, essentially as hypertextual documents.

It has been argued in several contexts that document management still lacks the flexibility offered by database systems. Therefore, a number of recent proposals aim at extending database-like techniques to textual data coming from the Web. In most cases, in fact, Web documents have a rather tight structure that can be assimilated to a database type. For example, a large number of organizations are now providing a Web access to their information systems and

* Work supported by Università di Roma Tre, MURST and Consiglio Nazionale delle Ricerche.

databases. Several commercial packages now support the activity of publishing in HTML format data coming from a DBMS. In this process, a collection of database objects — such as for example tuples corresponding to employees — are somehow "encoded" into HTML pages, to be browsed by users. Of course, these generated pages clearly reflect the regularities of the initial database.

In many cases, users do not have access to the original database: All they can access are the HTML files on the site. Therefore, in order to be able to apply database-like query techniques to Web pages, a key problem is to identify relevant pieces of information inside text, and to extract them in order to build an internal representation in some data model, which can then be manipulated by a query language. This process, which is usually referred to as a *wrapping* of a data source, can be seen as an attempt to "decode" the original database objects that have been encoded as HTML files for publishing purposes.

In this paper, we formally investigate exact methods for this *Schema Finding Problem* (SFP), which can be informally stated as follows: *Given a collection of strings encoding instances of some database type, derive the type and a database representation of the original instances.*

1.1 The Framework

To show a practical example of a schema finding problem, consider the Nagano Winter Olympics Web site [Nag]: The site contains data about athletes and events at the last Winter Olympics, and has been implemented using an object-relational database as a back-end [Las98].

Fig. 1. Athletes at the Winter Olympics

Coming from a database, pages in the site have a rather tight structure. For example, two pages from the site are shown in Figure 1; they report data about two athletes, Manuela di Centa and Ian Piccard. For each athlete, the name, the country, and a list of performances are reported[1]. Note that the site contains a page of this kind for each competing athlete. We can abstract the content of such pages as a counterpart to database objects of a nested type *athlete*, with attributes *name* and *country* of type *string*, plus an attribute *performances*, a collection of tuples, each reporting the *event, placement* and *performance*, as follows:

```
type athlete ( name:          string;
               country:       string;
               performances:  set ( event:        string;
                                    placement:    string;
                                    performance:  string; ) )
```

Given such a precise structure, it would be reasonable to think of using some high-level tool to ask complex queries, like *"Find all athletes in the site that won at least two medals"*. Unfortunately, we can't directly query the underlying database: the only way of accessing the site is by downloading raw-text HTML pages through the network and inspecting their content. We have no access to the original objects in the database, but rather to their *HTML-encoded versions*. This, of course, complicates the query process.

Note, in fact that whenever we download a page corresponding to a database object, the underlying structure is completely lost in favor of some HTML tagging. Before actually being able to query the site, we have somehow to re-discover the lost database schema, and, based on that, *decode* the original objects from the HTML strings, i.e., we face an instance of what we call the *Schema Finding Problem.*

1.2 Related Work

The early approaches to structuring and wrapping Web sites were essentially based on manual techniques [AM97,HGMC$^+$97,CM98]; these proposals assume that a human programmer examines a site and manually codes wrappers that abstract the logical features of HTML pages and store them in a local database. The focus, here, is on the development of languages and tools to support this wrapping process.

A number of other approaches have studied the problem of (semi-) automatically inferring the schema of a collection of fairly well structured Web pages. We briefly mention some in the following. Most of these approaches heavily rely on the use of heuristics. For example, in [AK97], the authors develop a practical approach to identify attributes in a HTML page; the technique is based on the identification of specific formatting tags (like the ones for headings, boldface, italics etc.) in order to recognize semantically relevant portions of a page.

[1] The page also contains several links, like, for example, a link to a biography, to the country page, and to each competition page. To simplify the discussion, we ignore the links here. In Section 4, we will discuss how to deal with attributes that are links to other pages.

An alternative approach is the one developed in [KWD97] and [Ade98]. In these cases, it is assumed that the wrapper generator has some a-priori knowledge about the semantics of a page. In [KWD97], *oracles* are used to this end. They are software modules embedding lexical knowledge about a particular application domain, used to recognize all occurrences of a relevant attribute in a page. In [Ade98] the user itself manually starts the semi-automatic structuring phase, by interactively labeling some example files, that is, by showing to the system some example attributes and their semantics. Then, based on this accumulated knowledge, in both cases a wrapper generator tries to infer the patterns according to which data items are organized in the target pages. This is accomplished by examining one page at a time, and progressively refining a grammar for the pages.

Similar problems have also been studied in other frameworks. For example, the work in [Bri98] aims at extracting sets of tuples of a pre-determined type (pairs of book titles and author names) not from a set of homogeneous pages in a site, but from possibly heterogeneous pages in the whole Web. In [NAM98] the authors address the problem of clustering similar objects in a semistructured Lore [ACC$^+$97] database, i.e., inferring some form of common schema starting from schemaless data.

The present problem presents similarities with the problem of identifying the common structure (in 2 dimensions) of a collection of objects that are available only as one-dimensional sequences which has gained a lot of attention recently for its applications in molecular biology [Wat89,HK93].

1.3 Contributions of the Paper

In this paper, we develop a formal framework for studying the Schema Finding Problem. Our approach departs from all the ones discussed above, in several respects. In fact, we aim at studying how far a completely automatic approach can be pushed in order to solve the schema finding problem; therefore, differently from [AK97], our approach makes very little use of heuristics; also, we do not assume any prior knowledge about the input data, as opposed to [KWD97,Ade98]. In fact, we are not primarily concerned with identifying the *semantics* of data in a page (i.e., the fact that a given string is the name of an athlete), but rather focus on the problem of recovering *types*. More specifically, given a set of encoded instances of a type T, we concentrate on the task of re-discovering the structure of type T. In this framework, a critical issue in recovering the scheme from a set of encoded instances is the ability of detecting collection (i.e., set) types from aggregation (i.e., tuple) types. Studying this relationship is a primary goal of the paper.

The techniques we use are also different from other proposals in the literature. In fact, given a collection of pages of a given logical type, instead of working on one page at a time, we try to infer the type by comparing different pages and looking at textual similarities between them. We formalize this process as a *decoding process*.

We identify nested relations as a natural abstraction for Web pages. This is motivated by several observations. First, HTML pages often have a nested structure, due to the presence of collections of items (like lists or tables); also, many Web sites are now built starting from relational and object-relational databases, and in some sense reflect the (possibly) nested structure of the underlying database. In fact, it has been shown in [AMM97,MMM98] that a data model based on nested relations is a promising abstraction for describing the structure of a large number of actual Web sites. Of course, this choice is by no means the most general one, and probably cannot capture all of the richness present in HTML pages. However, we feel it to be a good starting point for studying the schema finding problem.

To settle the background, in Section 2 we formalize the notion of encoding of database instances into sequences, and introduce various classes of encodings. We first consider a general version of the schema finding problem (SFP), where the encoding function is part of the input. In the general case, we show that the SFP is partial recursive if the encoding function is recursive. We then refine the class of encoding functions and relate the space complexity of the SFP to the time complexity of the encoding.

In practice, the encoding function is not given with the encoded data. It is therefore desirable to be able to solve the SFP without knowledge of the encoding function. We further restrict the encoding functions, and introduce the *mark-up encodings*, which satisfy the previous requirement, and, at the same time, nicely abstract SGML-like languages [ISO86] – like HTML – which are used in practice to encode information. Essentially, the mark-up encodings map database objects into strings in which the constants are encoded as strings over some alphabet, Δ, and the structure is encoded by arbitrary tags in some other alphabet, Σ. We distinguish between strict and loose mark-up, depending upon the encoding of empty sets. In the strict case, the structure of empty sets is maintained in the encoding, while it is lost in the loose case.

Then, in Section 3 we restrict our study to data that can be represented as nested relations and develop an algorithm that takes as input a set of strings and, by successive alignments, progressively refine their type. Our approach relies on a close correspondence between nested objects and regular expressions. In fact, we show that, given a collection of encoded tuples, finding a database type that subsumes them is reducible to finding a regular expression defining a language containing the input strings. Note that the containment problem for regular expressions is known to be PSPACE complete [Pap94]. However, dealing with restricted classes of expressions, our algorithms run in polynomial time.

The main contribution of the paper is the definition of two on-line algorithms for solving the schema finding problem in the case of mark-up encodings. The first algorithm works on strict mark-up, and uses a buffer of size linear in the size of an object (e.g. a tuple of a relation). The second algorithm works in the loose case and uses a buffer whose size is linear in the size of the loaded data. Both algorithms are shown to work in polynomial time.

Finally, in Section 4, we show how in practice, these algorithms can be used to load data from the Web into a database by first extracting the schema, and then wrapping HTML pages according to the schema. We also exhibit examples of HTML documents on which we illustrate the use of the proposed techniques. The algorithms are under implementation as part of a system for schema-finding on the Web. For space reasons, proofs of the theorems and code of the algorithms are omitted.

2 The Encoding of Data

In this section, we describe the abstract data to which we restrict our attention, and introduce formal methods for encoding it into sequences of characters. We also define our fundamental problem.

2.1 Abstract data

The abstract data we consider are nested relations [AB95,Hul88], that is typed objects allowing nested sets and tuples. The types are defined as follows. We assume the existence of an atomic type U, called the *basic type*, whose domain, denoted by $dom(U)$ is a countable set of constants. Other *types* (and their respective domains) can be recursively defined as follows: (i) If $T_1, \ldots, T_n$ are basic or set types, then $[T_1, \ldots, T_n]$ is a *tuple* type, with domain $dom([T_1, \ldots, T_n]) = \{[a_1, \ldots, a_n] \mid a_i \in dom(T_i)\}$; (ii) If T is a tuple type, then $\{T\}$ is a *set* type, with domain $dom(\{T\}) = \mathcal{P}_f(dom(T))$ (where $\mathcal{P}_f(S)$ denotes the collection of finite subsets of S).

Given a type T, an *instance* of T is an element of $dom(T)$. Let $\mathcal{I}$ be the set of all instances. A *relational instance* is an instance of a set type. Classical (flat) relations are instances of unnested set types, and nested relations are instances of arbitrary set types. Types can be conveniently represented as trees [Hul88], which can be constructed recursively as follows. (i) The basic type, U, is a leaf tree, U. (ii) A tuple type $[T_1, \ldots, T_n]$ corresponds to a tree rooted at a *tuple node*, with n subtrees, one for each T_i. (iii) A set type $\{T\}$ corresponds to a tree rooted at a *set node*, with one subtree. Similarly, instances can also be represented as trees, with constant instances, c, represented as a leaf tree c; tuple instances, $[t_1, \ldots, t_n]$, represented as a tree rooted at a *tuple node*, with n subtrees, one for each t_i; and set instances $\{t_1; \ldots; t_k\}$ represented as a tree rooted at a *set instance node*, with k subtrees, corresponding to the elements $t_1, \ldots, t_k$. If I is an instance, any subtree of I is a *subinstance* of I.

We introduce a labeling of type trees, recursively defined as follows. The root is labeled by the empty string ϵ. If a set node is labeled α, then its child is labeled $\alpha.0$; and if a tuple node with n children is labeled α, then its children are labeled $\alpha.1, \ldots, \alpha.n$. Instances are labeled similarly, with the children of a set node labeled α, all labeled $\alpha.0$. So an instance and its type have the same set of labels.

Instances can sometimes underuse their type. It has been shown in [GV95], that it has a strong impact on the expressive power of query languages. In the present context, we show that it is fundamental for finding the schema. An instance underuses its type for example when set types are used to model objects that always have the same cardinality (and would have been more accurately typed with a tuple), or when the attribute of a tuple always has the same value (and could have been omitted). We define this notion formally.

Definition 1. *A collection of instances I is* rich *with respect to a type σ if it satisfies the following two properties: (i)* set richness*: for each label α of a set node, there are sets of distinct cardinalities labeled α, and (ii)* tuple richness*: for each label α of a tuple node of arity k, and each $i \leq k$, there are distinct objects labeled $\alpha.i$*

Intuitively, if a collection of instances is rich with respect to a type, it makes full use of this type, and we will see in the sequel that it then contains enough information to recover the type.

2.2 Encoding of abstract data

We next study the encoding of instances into sequences of characters over a finite alphabet Σ. We define an *abstract encoding function*, *enc*, as a 1-1 mapping from the set of all instances $\mathcal{I}$ to Σ^*. We consider the problem of reversing an encoding, that is given a string in Σ^*, find the abstract data that it represents and its type.

For computation purposes, the abstract data is manipulated as strings obtained by a simple *standard encoding* based on a parenthetical representation, which is essentially the one used in the paper to "write" abstract objects. We consider an ordered alphabet Δ. The order on Δ induces a (lexicographic) order on Δ^*, which in turn induces an order on the set of all instances $\mathcal{I}$ ($I \leq J$ if $enc(I) \leq enc(J)$), and in particular on U. The standard encoding ξ of an instance I is defined recursively as follows. Constants are encoded by words in Δ^*. Tuples $[a_1, \ldots, a_n]$ by the string $[\xi(a_1), \ldots, \xi(a_n)]$, and sets $\{a_1, \ldots, a_n\}$, by the string $\{\xi(a_{i_1}), \ldots, \xi(a_{i_n})\}$, where $\xi(a_{i_1}) < \ldots < \xi(a_{i_n})$. Let $\bar{\mathcal{I}} \subset (\Delta \cup \{"\{", ",", "\}", "[", "]"\})^*$ be the set of standard representation of instances.

Concrete encoding functions are 1-1 mappings from $\bar{\mathcal{I}}$ to Σ^*. When it is clear from the context we simply say instance for either the abstract instance or its standard representation, and, similarly, encoding function for abstract or concrete encoding function. Let us now define formally the problem.

Definition 2. *Schema Finding Problem (SFP)*
Input*: An encoding function enc, and a finite collection W of strings of Σ^*.*
Output*: A type σ, and a collection C of standard representation of instances of type σ, such that $enc : C \to W$ is a bijection.*

We can immediately make the following general observation. If *enc* is a recursive function, then the schema finding problem is partial recursive. Indeed, it suffices to enumerate all possible instances, $I \in \bar{\mathcal{I}}$, and check if $enc(I) = w$ for some $w \in W$. If W is a collection of strings corresponding to instances, then it halts when all such instances have been enumerated.

We now consider the complexity of the schema finding problem in the size of the collection W. The size $|w|$ of a sequence w is defined as usual as the length of w. The *size* $|I|$ of an instance I is defined as the size of its standard encoding as a sequence [AHV94]. The *domain* of an instance is the set of constants occurring in the instance, and for the standard encoding of the instance, the domain is the set of words in Δ^* encoding the constants.

It is not sufficient to restrict the complexity of encoding functions in order to bound the complexity of the schema finding problem. We next propose a sufficient restriction on the encodings to ensure a complexity bound on the SFP. We say that an encoding *enc* is *non-compacting*, if there exists a constant C such that for any instance I, $|I| \leq C|enc(I)|$. We can now give the correlation between the complexity of the encoding and the complexity of the SFP.

Theorem 1. *Let enc be a non-compacting encoding function having complexity* $\text{TIME}(f(n))$, *($f(n) \geq n$). Then the SFP can be solved in* $\text{SPACE}(f(n))$.

The previous result provides sufficient conditions for bounding the complexity of the schema finding problem. These conditions do not ensure tractability though. Moreover, we would like to restrict further the class of encodings, to obtain an encoding–invariant solution to the SFP. This is a desirable property, since in practice, encoding functions are not given, and only the encoded data can be found. So our approach is to find classes of encoding functions such that the SFP can be decided without the knowledge of the encoding function. Note that this is a strong assumption, which implies that the result does not depend upon the chosen encoding. For that purpose we introduce a restricted class of non-compacting encodings, namely *mark-up encodings*.

2.3 Mark-up encodings

In a mark-up encoding the schema and the data are encoded in disjoint finite alphabets, the *schema alphabet*, Σ, and the data alphabet, Δ. As for the standard encoding, constants are encoded as words in Δ^+, by some *data encoding*, δ, that is a 1-1 mapping from U to Δ^+.

Consider a type σ, and a corresponding labeled tree T_σ. We define a *tagging function* as a function that associates a pair of strings, called *start* and *end tags* to each label in $\mathcal{L}$, in such a way that respectively basic, set and tuple labels have different start and end tags. More formally, let Θ be a prefix subset[2] of Σ^*. Let $\Theta_1 \cup \Theta_2 \cup \Theta_3$ be a partition of Θ. A *tagging function* is a mapping from the set of labels $\mathcal{L}$ of T_σ to $\Theta \times \Theta$, which associates to each basic (resp. tuple, set) label $\alpha \in \mathcal{L}$ a pair of strings in Θ_1 (resp. Θ_2, Θ_3), denoted by $start(\alpha)$ and $end(\alpha)$, with $start(\alpha) \neq end(\alpha)$.

[2] Recall that a set of strings Θ is called a prefix set if no word of Θ is a prefix of another word of Θ [CR94].

Definition 3. *A* (loose) mark-up encoding *enc based on a structure* $(\Sigma, \Delta, <_\Delta, \delta, tag)$ *— where* Σ *and* Δ *are two disjoint finite alphabets,* $<_\Delta$ *is an order on* Δ*,* δ *a data encoding, and tag a tagging function — is a function recursively defined on the tree of an instance as follows:*

- *for a leaf node* $u \in U$ *with label* α*,* $enc(u) = start(\alpha) \cdot \delta(u) \cdot end(\alpha)$*;*
- *for a tuple node* $[T_1, \ldots, T_n]$ *with label* α*,* $enc([T_1, \ldots, T_n]) = start(\alpha) \cdot enc(T_1) \cdot \ldots \cdot enc(T_n) \cdot end(\alpha)$*;*
- *for a non-empty set-instance node* $\{T_1; \ldots; T_n\}$ *of label* α*,* $enc(\{T_1; \ldots; T_n\}) = start(\alpha) \cdot enc(T_{i_1}) \cdot \ldots \cdot enc(T_{i_n}) \cdot end(\alpha)$*, where set elements have been recursively ordered according to* $<_\Delta$*; empty set-instances are encoded by* $start(\alpha) \cdot end(\alpha)$*.*

We distinguish between *loose* and *strict* mark-up encodings, based on the encoding of empty sets. Suppose U contains a null value *null*. Given a set type σ, we define its *null instance*, $null(\sigma)$ as an instance whose tree is obtained from σ by replacing each leaf node U with a leaf node *null*. A *strict mark-up encoding* is defined as a loose mark-up encoding with the following difference: the encoding of an instance I is the encoding of instance I' obtained by replacing in I each empty set-instance of a type σ by $null(\sigma)$.

The definition of loose (strict) mark-up encoding can be extended also to types. In this case, function *enc* associates a *regular expression* $enc(\sigma)$ over $(\Sigma \cup \Delta)^*$ with each type σ as follows:

- for a leaf node U labeled α, $enc(U) = start(\alpha) \cdot (\Delta)^+ \cdot end(\alpha)$;
- a tuple node is encoded as above;
- for a set node $\{T\}$ with label α, $enc(\{T\}) = start(\alpha) \cdot (enc(T))^* \cdot end(\alpha)$; ($enc(\{T\}) = start(\alpha) \cdot (enc(T))^+ \cdot end(\alpha)$ for strict encoding);

Since the mark-up encoding of an instance is a string in $(\Sigma \cup \Delta)^*$, and the mark-up encoding of a type is a regular expression over $(\Sigma \cup \Delta)^*$, we can establish a close correspondence with the theory of regular languages; in fact, the class of mark-up encodings has the following property. As usual, we note $L(exp)$, the language defined by a regular expression *exp*.

Proposition 1. *Given a (strict/loose) mark-up encoding enc based on a structure* $(\Sigma, \Delta, <_\Delta, \delta, tag)$*, then, for each instance* I *of type* σ*,* $enc(I) \in L(enc(\sigma))$*.*

Figure 2 shows several types and instances with their corresponding encodings with $\Delta = \{1, 2, 3, \mathbf{NULL}\}$, and Σ containing any character not in Δ. Note that empty sets are encoded with null values in the strict mark-up encoding, while they are simply omitted in the loose mark-up encoding.

For a collection of instances $\mathcal{J}$, we denote by $enc(\mathcal{J})$, the collection of encoded instances of $\mathcal{J}$. The class of mark-up encodings verify the following fundamental property, which states that the choice of a specific encoding is irrelevant.

Theorem 2. *For all mark-up encodings* enc, enc' *over the same* Σ, Δ, δ*, and all set-rich collections of instances* $\mathcal{J}, \mathcal{J}'$ *of type* σ*, if* $enc(\mathcal{J}) = enc'(\mathcal{J}')$*, then* $\mathcal{J} = \mathcal{J}'$*.*

σ_1 = [U, {[U, U]}]
strict mark-up encoding <t>eΔ^+/e<s>(<t>eΔ^+/eeΔ^+/e</t>)$^+$</s></t>
loose mark-up encoding <t>eΔ^+/e<s>(<t>eΔ^+/eeΔ^+/e</t>)*</s></t>
σ_2 = [U, {[U]}]
strict mark-up encoding romaregesΔ^+unl(aeΔ^+iacta)$^+$tpqr
loose mark-up encoding romaregesΔ^+unl(aeΔ^+iacta)*tpqr
I_1 = [1, {}] *of type* σ_1
strict mark-up encoding <t>e1/e<s><t>e**NULL**/ee**NULL**/e</t></s></t>
loose mark-up encoding <t>e1/e<s></s></t>
I_2 = [1, {[2]; [3]}] *of type* σ_2
strict mark-up encoding romareges**1**1unlae**2**2iactaae**3**3iactatpqr
loose mark-up encoding same as strict

Fig. 2. Examples of encodings

The proof is made by induction on the nesting of the instances. Theorem 2 has the fundamental consequence, that the decoding can be done without knowledge of the encoding function. In the next section, we investigate the SFP in this context.

3 Schema Finding for Mark-Up Encodings

In this section, we introduce several algorithms to solve the SFP problem assuming mark-up encodings. We first restate the problem in this special framework:

Definition 4. *Schema Finding Problem for Mark-up Encodings*
Input: *Σ, Δ, δ as defined above, and a finite collection W of strings of $(\Sigma \cup \Delta)^*$.*
Output: *A type σ, and a collection C of instances of type σ, such that there is a mark-up encoding enc such that $enc : C \to W$ is a bijection.*

For simplicity, we focus in the sequel on inputs consisting of a finite collection of encodings $enc(I_1)$, $enc(I_2)$, ..., $enc(I_n)$ of instances I_1, I_2, ..., I_n of a nested tuple type σ, according to some mark-up encoding function *enc*. Our algorithms work on encoded instances, and progressively construct the type by generating *templates*. Templates generalize both types and instances, and represent partially specified types. Together with templates, we introduce a reflexive *subsumption* relation between templates, denoted by $\preceq$, which extends the relationship between a type and its instances. $T =$`[c, {[U]}]` and $T' =$`[U, {[a, U]; [a, U]; [a, U]}]` are examples of templates. It is easy to see that they correspond to partially specified types. More specifically, T subsumes any tuple of two attributes, the first one being a `c`, and the second one any set of monadic tuples. T' subsumes any tuple of two attributes, the second one being a set of exactly three binary tuples, having `a` as first attribute. Types and instances of Figure 2 provide more examples of templates, satisfying the relation: type σ_1 subsumes T', and σ_2 subsumes T. Templates and the subsumption relation can be formally defined as follows:

- Every element $u \in U$ is a *constant* template; a constant template subsumes itself.
- U, the basic type, is a *basic* template; the basic template, U, subsumes every template in $dom(U)$.
- If $T_1, \ldots, T_n$ are templates, then $[T_1, \ldots, T_n]$ is a *tuple* template; a tuple template, $[T_1, \ldots, T_n]$ subsumes any template in $\{[t_1, \ldots, t_n] \mid t_i \preceq T_i)\}$.
- If T is a template, then $\{T\}$ is a *set* template; a set template, $\{T\}$, subsumes any template in $\mathcal{P}_f(\{t \mid t \preceq T\})$.
- If $T_1, \ldots, T_k$ are templates, then $T = \{T_1; \ldots; T_k\}$ is a *set instance* template; $k \geq 0$ is called the *cardinality* of T; a set instance template subsumes any template in $\{\{a_1; \ldots; a_k\} \mid a_i \preceq T_i\}$.

It is easy to see that: (i) every type is a template, constructed using only tuple, set and basic sub-templates; (ii) every instance of a type is itself a template, made of tuple, set-instance and constant sub-templates. In the following, we blur the distinction between a types or an instance and the corresponding template. Note also that each template corresponding to a type subsumes all of its instances.

We denote by $\mathcal{T}$ the universe of all templates. The relation $\preceq$ defines a partial order on the set $\mathcal{T}$.[3] We say that two templates T_1, T_2 are *homogeneous* if they admit a common ancestor, that is, there exists a template $T \in \mathcal{T}$ such that $T_1 \preceq T$ and $T_2 \preceq T$. Intuitively, two templates are homogeneous if they represent objects that are subsumed by the same type. It is now easy to see that, for each maximal set of homogeneous templates, $\mathcal{T}_H$, $(\mathcal{T}_H, \preceq)$ is a join-semilattice. Given a template T, we denote by $\mathcal{H}(T)$ the class of all templates homogenous to T. In the following, unless explicitly specified, we will always refer to join-semilattices of homogeneous templates. Given a finite collection S of homogeneous templates, $\text{LUB}(S)$ will denote the least upper bound of elements in S in the corresponding join-semilattice.

Consider the relationship between a type and its instances. Since a type subsumes all of its instances, given a set of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ of some type σ, σ is a common upper bound of $\mathcal{I}$ in the join-semilattice of templates homogeneous to σ, $\mathcal{H}(\sigma)$. The following proposition shows that, if $\mathcal{I}$ makes full

[3] Similar orderings for database types have been used also in other frameworks; see, for example, [Oho90].

use of its type according to the definition given in Section 2, then this upper bound is the least one.

Theorem 3. *Given a set of instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ of type σ, $\mathcal{I}$ is rich for σ iff $\sigma = \text{LUB}(\mathcal{I})$.*

This result shows that solving the schema finding problem amounts to computing the least upper bound of a set of instances according to the template subsumption relation. However, we are not dealing with abstract objects, but with strings that encode them. We next show how to solve the problem using regular expressions.

First, note that the tree representation of types and instances, the tree labeling function and the notion of mark-up encoding introduced in Section 2 extend immediately to templates. There is a close relationship between template subsumption, $\preceq$, and the familiar concept of containment, $\subseteq$, between regular expressions, as stated by the following theorem.

Theorem 4. *Given a set-rich collection $\mathcal{I} = \{I_1, \ldots I_n\}$ of instances of a type σ, and a mark-up encoding enc, let us call $enc(\mathcal{H}(\sigma))$ the image of $\mathcal{H}(\sigma)$ according to enc. Then, $(enc(\mathcal{H}(\sigma)), \subseteq)$ is a join-semilattice, and: $\text{LUB}_{\subseteq}(enc(I_1), \ldots, enc(I_n)) = enc(\text{LUB}_{\preceq}(I_1, \ldots, I_n))$*

Another important fact is the following:

Proposition 2. *Given an encoding $enc(T)$ of a set-instance free template T, it is possible to derive T from $enc(T)$ in linear time.*

Theorem 4 and Proposition 2 suggest that we can solve the schema finding problem working on strings and join-semilattices of regular expressions.[4] Given a set of encoded instances of a nested tuple type σ, $\mathcal{E} = \{e_1, \ldots, e_n\}$ according to some mark-up encoding, the strategy to solve the schema finding problem is: (i) to find the regular expression encoding σ as the least upper bound $e_\sigma = \text{LUB}_{\subseteq}(\mathcal{E})$; (ii) from the regular expression, to construct σ; (iii) based on the grammar defined by e_σ, from each e_j to derive a representation of an instance I_j of σ.

3.1 Schema finding with strict mark-up

Let us first consider the case in which inputs are encoded using a *strict* mark-up encoding function. Recall that, for every join-semilattice, the least upper bound operator is associative, and therefore, given a set of elements, $\mathcal{E}$, we can progressively compute the least upper bound of the set, independently of the order $e_1, \ldots e_k$ for elements in $\mathcal{E}$, based on the following iterative algorithm.

[4] With respect to Theorem 4, note that the two join-semilattices, $(\mathcal{H}(\sigma), \preceq)$ and $(enc(\mathcal{H}(\sigma)), \subseteq)$, are not in general isomorphic. This is due to the fact that the notion of subsumption between set and set-instance templates is essentially order-independent, whereas containment between regular expressions depends on the actual order according to which elements in set instances are listed.

$$\begin{cases} lub_1 & = e_1 \\ lub_{i+1} & = \textsc{lub}(lub_i, e_{i+1}), \quad \text{for } i = 2, \ldots, k \end{cases}$$

Equations above suggest an efficient on-line strategy to solve the schema finding problem with mark-up encoding, assuming we know how to compute least upper bounds of regular expressions. Note that computing upper bounds of regular expression implies testing containment. The containment problem for regular expression is complete for PSPACE [Pap94]. However, in this context, we deal with rather simplified regular expressions, for which we prove that the containment problem is in PTIME. This is due to the very limited use of union – essentially only as a part of Δ^+ – and to the fact that subexpressions are clearly marked using tags.

We have developed an algorithm, called `matchSchema`, that computes the least upper bound of two expressions of this sort in polynomial time, as follows:

Theorem 5. *Given two regular expressions e_1, e_2 corresponding to encodings of homogeneous templates $T_1, T_2 \in \mathcal{T}_H$, according to encoding enc, let $n = max(|e_1|, |e_2|)$. Algorithm* `matchSchema` *computes $\textsc{lub}(e_1, e_2) \in enc(\mathcal{T}_H)$ in time $\mathcal{O}(n^2 log(n))$.*

Algorithm `matchSchema` tries to align the two input expressions. In presence of mismatches, it attempts to dynamically change the inputs, and then recursively align again. Every time an input is changed, it is transformed into a new regular expression, corresponding to a template that subsumes the previous one. The algorithm terminates when, by successive changes, the two input have been transformed into a common regular expression, which is the least upper bound.

The procedure `findSets` is the core of the algorithm. It tries to find repeated contiguous patterns of the form w^n corresponding to elements of set instances inside a regular expression, and replace them by $(w)^+$. This process is far from being a trivial task, and implies a number of subtleties. First, the procedure is recursive, since sets may contain nested sets. Second, not all contiguous squares actually correspond to sets of homogeneous elements. To see this, consider the following tuple: `[a{[b{[b...]`; here, square `{[b` is due to a repetition of structure at different levels, and not to instances of a set.

Note that, in the case the input collection of encoded instances is not rich for its type, the on-line algorithm computes a regular expression e which corresponds to a template subsumed by σ, that is, to a partially specified type. However, if the input is set-rich, we can recover the schema.

Proposition 3. *Given a set of regular expressions $\mathcal{E} = \{e_1, \ldots, e_n\}$ corresponding to instances of some type σ according to mark-up encoding enc. If $\mathcal{E}$ is set-rich with respect to σ, then: $e = \textsc{lub}(e_1, \ldots, e_n) \preceq enc(\sigma)$ and σ can be built from e in linear time wrt the size of e.*

The proof follows from the fact that, since $\mathcal{E}$ is set-rich, e is set-instance free, and therefore, by Proposition 2, σ is uniquely identified. Proposition 3 is very useful in a number of practical cases, as discussed in Section 4. Based on `matchSchema`, we can define algorithm `schemaFinding1` to solve the SFP for strict mark-up encodings as follows: the algorithm progressively loads the encoded tuples and compares them; it first applies `matchSchema` on the first two tuples and then iteratively applies `matchSchema` on the previously obtained regular expression and each new tuple. Algorithm `schemaFinding1` works on-line, using a buffer of size bounded by the size of the longest encoded tuple. Since `matchSchema` runs in polynomial time, the overall computation is polynomial.

3.2 Schema finding with loose mark-up

Consider now the schema finding problem in the case of loose mark-up encodings. In this case, empty sets are encoded simply by the start and end tags, so that their inner structure is not visible. Algorithm `matchSchema` fails on instances containing empty sets under loose encoding.

We define the notion of *empty-set free template* as a template T such that all subtrees of T corresponding to set-instance templates have cardinality greater than 0. A template is said to have *empty sets* otherwise. The following proposition justifies the use of Algorithm `matchSchema` on loose mark-up encodings.

Proposition 4. *Given a loose mark-up encoding function, enc, and two regular expressions, e_1, e_2 corresponding to encodings of homogeneous templates T_1, T_2, call e' the output of* `matchSchema` *on e_1, e_2; then: (i) if both T_1, T_2 are empty-set free templates, then $e' = \mathrm{LUB}(T_1, T_2)$; (ii) if only one of T_1, T_2 is empty-set free, then $e' = null$.*

Based on Proposition 4, we can develop a polynomial on-line strategy for solving the schema-finding problem with loose mark-up encoding, with a slightly more demanding notion of richness, and a larger buffer. We say that a set of instances $\mathcal{I}$ of a type σ is *strongly rich* for σ if the subset of $\mathcal{I}$ of empty-set free instances is rich for σ. Intuitively, as far as sets are concerned, this corresponds to requiring that, for each set subtree of σ, there are at least two different non-empty cardinalities.

In this case, an on-line algorithm – let us call it `schemaFinding2` – for solving the schema-finding problem in case of loose mark-up encoding can be informally described as follows; suppose $e_1, \ldots, e_n$ denote the input:

1. `schemaFinding2` works on a data structure which is an array of templates, `Templ`, which initially contains e_1;
2. each e_i, $i = 2, \ldots, n$ is sequentially compared with templates in `Templ` using Algorithm `matchSchema`; we say that an e_i matches a template T if `matchSchema(`T`,` e_i`)` is not null; if a matching template is found, the match is recorded and the next instance is examined; if no matching template is found in `Templ`, e_i is added to `Templ` as a new template;
3. at the end, several regular expressions will be generated in `Templ`; if the input was strongly rich, there is at least one expression e_σ in which all sets have been marked; a simple scan of `Templ` allows to identify e_σ, since it is the expression with the maximum number of marked sets (corresponding to +-subexpressions); by replacing each + with * in e_σ, we have a grammar for parsing encoded instances and build representatives.

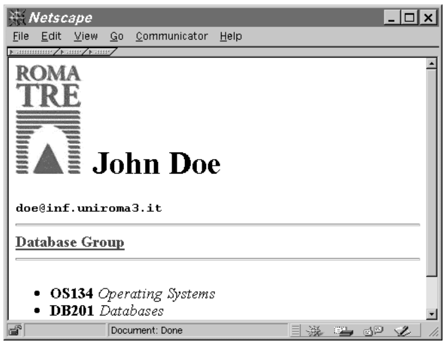
The difference between algorithms `schemaFinding1` and `schemaFinding2` is that the latter has a buffer-size bounded by the size of the loaded tuples (versus the size of a single tuple) and its running time is bounded by a polynomial in the size of the loaded tuples.

4 Schema Finding on the Web

Frequently, Web sites have a rather tight structure; for example, a large number of organizations are now providing a Web access to their information systems and databases. Pages are generated on-the-fly by executing queries on the database and returning results in HTML format, which clearly reflect the regularities in the initial database. However, schema finding techniques are needed to rediscover such regularities in files.

We claim that the techniques presented in this paper, enriched by simple heuristics, can effectively support schema finding on the Web. The notion of mark-up encoding is in fact inspired by SGML-like mark-up languages, of which HTML is an example. In these languages, *tags* – i.e., clearly identifiable pieces of code – are used to mark data inside the document, with two main goals: defining the logical role of a piece of information (like titles, paragraphs, lists etc.) and contribute to the layout on the browser. Consider for example a university department Web site which has been generated automatically starting from a database, a case that is becoming rather frequent. Suppose, for example, the database contains a nested relation *Professor* $\{[$*Name, email, ResearchGroup, Courses:* $\{[$*Code, Title*$]\}]\}$, and that for each professor, i.e., for each tuple in the relation, an HTML page like the following has been generated:

```
<HTML><BODY BGCOLOR="FFFFFF">
<H1><IMG SRC="deptsymbol.gif">
      John Doe </H1>
<TT> doe@inf.uniroma3.it </TT>
<HR><A HREF="db.html">
      Database Group </A>
<HR><UL>
  <LI><B> OS134 </B>
      <I> Operating Systems </I>
  <LI><B> DB201 </B>
      <I> Databases </I>
  <LI>...
</UL></BODY> </HTML>
```



These HTML pages can be easily turned into mark-up encodings of the original tuples. In fact, roughly speaking, tags are used in the page to encode the original database structure, and strings to encode data. First, constants in the database are strings, i.e, words in the latin alphabet (let us denote it Σ_{latin}) and are therefore encoded by themselves, that is, function δ is the identity function. Second, we may use a simple pre-processing of HTML code to distinguish tags from other strings and replace the former with special tokens from an alphabet

Σ_{tag}. Note that this has to be done with special care, since URLs, i.e., references to other pages, are embedded inside tags `<A>` and `</A>`; since URLs are to be considered as meaningful attributes of a page, the tokenization process should extract URLs from `<A>` tags and treat them as strings in the data alphabet, Σ_{latin}. A similar treatment is needed for images. In this way, pages in our departmental site are transformed into strings over $(\Sigma_{latin} \cup \Sigma_{tag})^*$ which are actually mark-up encodings, according to the definition given in Section 2.

Now, simple heuristics can be used to identify collections of pages with homogeneous structure inside the site; we will get to this later on in this section; for now, suppose that in the department site there is a page containing the list of all professors, with links to their respective pages, which can thus be easily identified. We can run our algorithms on such collection of pre-processed pages, and derive the following grammar, from which the nested schema is easily found:

<HTML> <BODY BGCOLOR="FFFFFF">
<H1><IMG SRC="deptsymbol.gif"> Σ^*_{latin} </H1><HR>
<TT> Σ^*_{latin} </TT> <A HREF=" Σ^*_{latin} "> Σ^*_{latin} </A>
<HR><UL> (<LI><B> Σ^*_{latin} </B> <I> Σ^*_{latin} </I>)* </UL>
</BODY> </HTML>

After this schema has been derived, a human operator can examine the schema and – by looking at actual pages on the site – associate a semantics to each attribute, allowing in this way to tell names from e-mails. Note also that a parser for the grammar is a natural wrapper for the pages.

It might be argued that the example above has a rather strong separation between data and structure in the page. Indeed, in many cases, this separation will be less strict. In fact, to improve readability, often the *role* of data inside a page is marked not only using tags, but also by means of attribute names, comments, remarks. With respect to the HTML page above, beside tags, also constant strings – like "e-mail", "Research Group", "List of courses" – could be used in the encoding. The presence of these "metadata" can somehow complicate the schema finding process, since pages will contain pieces of information that are not in the original database. Interestingly, the techniques developed in Section 3 allow to deal correctly with these situations. In fact, our algorithms will detect that all pages of professors contain the same constant strings, which therefore represent metadata and can be excluded from the schema. (These strings common to all pages can be nicely used instead to automatically suggest a semantics for attributes.)

As mentioned above, another important issue in this context consists in finding groups of homogeneous pages in a site, which can then be processed by the algorithms. There are several empirical ways to do this: for example, by taking collections of pages inside the same physical directory, or all pages linked to the same list. Also, several tools exist that navigate a site and produce a map of the site content, thus giving a graphical representation of the site topology, which may help in identifying homogeneous groups of pages. Let us briefly mention that, under the reasonable assumption that pages containing different information do have different features, our algorithms can be easily extended in such a way as to load the whole site, compare pages and cluster them in different types. In fact, it is easy to see that, whenever two pages of different types, containing perhaps different tags are compared, the algorithms returns a null common schema, thus identifying the mismatch.

5 Conclusion

The algorithms have only been tested on a few samples yet. Nevertheless, we are rather optimistic. Indeed, the number of tuples that are necessary to recover the schema of a nested relation seems very low. Consider random instances with a probability p that, for a given label α, two instances have all sets with label α of equal cardinality. Then, the probability that a collection of n instances has all sets with label α of the same cardinality is $(1 - p^{n-1})$. Assuming that the probabilities are independent for different labels, then, the probability, that a collection of n instances with k set labels is set-rich, is $(1 - p^{n-1})^k$. So, for example if $p = \frac{1}{2}$, the probability that the schema of a relation with 5 sets in its type is found after looking at 10 tuples is 99%! The previous observation leads to a linear time algorithm for extracting the schema and loading the tuples in the right format. Indeed, once the schema is obtained (in an amount of time which in average depends only upon the schema size), checking that a tuple is of the right form, and loading it can be done in linear time.

In the present paper, we have concentrated on abstract data consisting of nested relations. A natural and useful extension would be to allow union types and therefore heterogeneous sets to model semi-structured data [Abi97] which, in this context, seems fundamental. The algorithms developed here can be adapted to nested heterogeneous sets with the following outcome. We were able to extract a type which is subsumed by the original nested type with union, but we have no formal results yet in this case.

We are now implementing our algorithms as part of a system for schema-finding on the Web. The system downloads HTML pages from a target Web site and tries to infer their common structure. Although the techniques we propose are far from being a final solution to this problem, our experiences tell us that they represent a promising starting point. In fact, in all cases in which pages do comply with a tight structure, our algorithms have been able to correctly recover the type. Also, we show that in average only a small number of tuples is needed to identify the schema, thus providing a linear behavior of the method. We are now working to extend the framework in order to handle pages with "exceptions" [CM98], i.e., pages containing features that slightly depart from the common type.

Acknowledgments The authors would like to thank Pino Di Battista for helpful comments on on-line algorithms and Valter Crescenzi, who helped in the implementation of the algorithms.

References

[AB95] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. *The VLDB Journal*, 4(4):117–138, 1995.

[Abi97] S. Abiteboul. Querying semi-structured data. In *ICDT'97*.

[ACC+97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *Journal of Digital Libraries*, 1(1):5–19, April 1997.

[Ade98] B. Adelberg. NoDoSE – a tool for semi-automatically extracting structured and semistructured data from text documents. In *SIGMOD'98*.

[AHV94] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1994.

[AK97] N. Ashish and C. Knoblock. Wrapper generation for semistructured Internet sources. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with SIGMOD'97)*.

[AM97] P. Atzeni and G. Mecca. Cut and Paste. In *PODS'97*.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *VLDB'97*.

[Bri98] D. Brin. Extracting patterns and relations from the World Wide Web. In *Proceedings of the Workshop on the Web and Databases (WebDB'98) (in conjunction with EDBT'98)*.

[CM98] V. Crescenzi and G. Mecca. Grammars have exceptions. *Information Systems*, 1998. Special Issue on Semistructured Data, to appear.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[GV95] S. Grumbach and V. Vianu. Tractable query languages for complex object databases. *Journal of Computer and System Sciences*, 51(2):149–167, 1995.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the Web. In *Proceedings of the Workshop on the Management of Semistructured Data (in conjunction with ACM SIGMOD, 1997)*.

[HK93] K. Han and H. J. Kim. Prediction of common folding structures of homologous RNAs. *Nucleic Acids Research*, 21(5):1251–1257, 1993.

[Hul88] R. Hull. A survey of theoretical research on typed complex database objects. In J. Paredaens, editor, *Databases*, pages 193–256. Academic Press, 1988.

[ISO86] ISO. International Organization for Standardization. ISO-8879: Information Processing – Text and Office Systems - Standard Generalized Markup Language (SGML), October 1986.

[KWD97] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence (IJCAI'97)*, 1997.

[Las98] E. R. Lassettre. Olympic records for data at the 1998 Nagano Games. In *SIGMOD'98*. Industrial Session.

[MMM98] G. Mecca, A. Mendelzon, and P. Merialdo. Efficient queries over Web views. In *EDBT'98*.

[Nag] Nagano 1998 Winter Olympics Web site. `http://www.nagano.olympic.org`.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *SIGMOD*, 1998.

[Oho90] A. Ohori. Semantics of types for database objects. *Theoretical Computer Science*, 76(1):53–91, 1990.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Wat89] M.S. Waterman. *Mathematical Methods for DNA Sequences.* CRC Press, 1989.