Representative Objects: Concise Representations of Semistructured, Hierarchical Data

Svetlozar Nestorov, Jeffrey Ullman, Janet Wiener, Sudarshan Chawathe
Department of Computer Science
Stanford University
Stanford, CA 94305-9040, USA
{evtimov,ullman,wiener,chaw}@db.stanford.edu
http://www-db.stanford.edu

Abstract

In this paper we introduce the representative object*, which uncovers the inherent schema(s) in semistructured, hierarchical data sources and provides a concise description of the structure of the data. Semistructured data, unlike data stored in typical relational or object-oriented databases, does not have fixed schema that is known in advance and stored separately from the data. With the rapid growth of the World Wide Web, semistructured hierarchical data sources are becoming widely available to the casual user. The lack of external schema information currently makes browsing and querying these data sources inefficient at best, and impossible at worst. We show how* representative objects *make schema discovery efficient and facilitate the generation of meaningful queries over the data.*

1. Introduction

The goal of this paper is to present a tool, the "representative object," that facilitates querying and browsing of semistructured, hierarchical information, such as that found on the web. The lack of external schema information currently makes browsing and querying these 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 [6].

In this paper we introduce the *representative object* concept. The representative object allows browsers to uncover the inherent schema(s) in semistructured, hierarchical data. Representative objects are implemented in the *Lore* DBMS as "DataGuides" [5]. 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.

1.1. Motivating applications

- Schema discovery: To formulate any meaningful query for a semistructured, hierarchical 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, hierarchical data, we often need to express paths through the hierarchy 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.

1.2. Paper organization

In Section 2, we introduce our data model and define several terms and functions regarding the hierarchical and semistructured nature of the data, including the OEM (object-exchange model) used in the Tsimmis project at Stanford. Then in Section 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 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, we present a method based on determinization and minimization of nondeterministic finite automata for construction of a minimal FRO in OEM. Section 6 describes the construction and use of the simplest k-RO, the case $k = 1$. Sections 7 and 8 present two alternative approaches to building a k-RO for $k > 1$, a graph-based approach and an automaton-based approach. Section 9 presents the conclusions and outlines the future work.

2. Preliminaries

In this section we describe the data model used in the paper. The object-exchange model (OEM) [1] is designed specifically for representing semistructured data for which the representative objects are most applicable and useful. The OEM described in [1] that we use is a modification of the original OEM introduced in [4]. 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.

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).

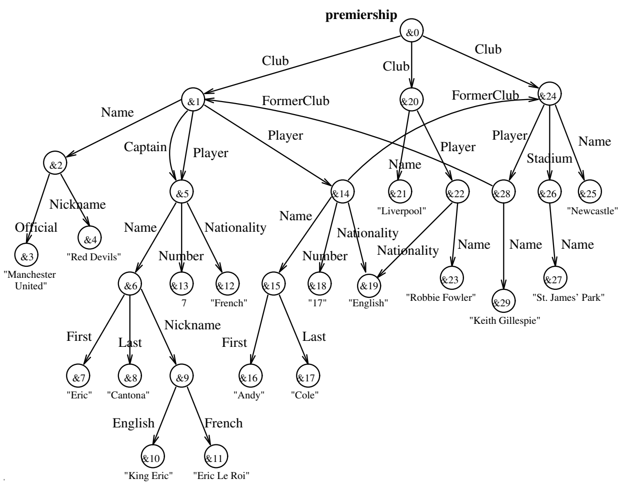

Figure 1. The premiership object.

Figure 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.

2.2. Simple path expressions and data paths

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

Definition 2.1 *Let l_i be a label (of object references) for $i = 1..n, n \geq 0$. Then $pe = l_1.l_2 \cdots l_n$ is a* simple path expression *of length n.*

Definition 2.2 *Let o_i be an object for $i = 0..n$, l_i be a label for $i = 1..n$, and $\langle l_i, identifier(o_i)\rangle \in value(o_{i-1})$ for $i = 1..n, n \geq 0$. 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 2.3 *Note that we allow data paths of length 0 that consist of a single object. We also allow a simple path expression of length 0. This simple path expression contains no labels and is denoted by the special symbol ϵ. Any data path of length 0 is an instance of ϵ.*

Example 2.4 *To illustrate the above terms consider the premiership object from Figure 1.*

- *The simple path expression* Player.Number *has two* instance data paths *within the premiership object, namely* obj(&1),Player,obj(&5),Number,obj(&13) *and* obj(&1),Player,obj(&14),Number,obj(&18).
- *Consider the following two data paths* obj(&1),Player,obj(&14),FormerClub,obj(&24), obj(&24),Player,obj(&28),FormerClub,obj(&1). *Thus,* obj(&1) *is within* obj(&24) *and* obj(&24) *is within* obj(&1), *i.e., there is a cycle within the premiership object.*

2.3. Continuations

The 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 represented in OEM. We briefly describe the schema discovery process before we give the formal definitions of the continuation functions.

Consider an object o 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 2.5 *Let o be an object in OEM and $pe = l_1.l_2 \cdots l_n$ a simple path expression, $n \geq 0$. Then 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*$\}$.

If we view OEM as a graph Definition 2.5 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 i-th step, $0 \leq i \leq n$, we pick an edge labeled with the i-th label in pe. At the last, $n + 1$-th step we pick any edge. Then the continuation of o and pe is the set of all labels on the edges we picked at the last step of a traversal described above plus $\perp$ if in any of the traversals we made the first n steps but could not make the $n + 1$ step because we ended up in a vertex with no outgoing edges (corresponding to an atomic object).

Example 2.6 *Consider the premiership object from Figure 1. The following examples illustrate Definition 2.5.*

- $continuation(premiership, \epsilon) = \{Club\}$
- $continuation(premiership,$ *Club*$)$ = $\{$*Name, Player, Stadium,*
- $continuation(premiership,$ *Club.Player.Name*$)$ = $\{$*First, Last, Nickname,* $\perp\}$

Note that in Definition 2.5 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 2.7 *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*$\}$.

- *Otherwise (if* $n < k$*)* $continuation^k(o, pe) = continuation(o, pe)$.

Example 2.8 *Consider the premiership object in Figure 1. The following examples illustrate Definition 2.7.*

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

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

Lemma 2.9 *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 2.7. 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 2.5 is a subset of the set of data paths considered in Definition 2.7. 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 2.7 only the data paths rooted at o are effectively considered which is the the set of data paths considered in Definition 2.5.

3. Representative object definitions

A "representative object" for an object o in OEM is any implementation of the $continuation$ function for o. We refer to these implementations as "representative objects" because in fact they are implemented in practice as objects in OEM. However, as discussed in later sections of this paper, there are many different ways to represent the $continuation$ function, and not all are "objects" in the usual sense. For instance, we discuss graph-based and 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 1.1. We then 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.

3.1. Full representative objects

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

Definition 3.1 *Let* o *be an 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 1.1.

3.1.1 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 1. This approach to exploration has been implemented in the *DataGuide* feature of Lore, a database system using the OEM, as discussed in [5].

Example 3.2 *Suppose we start at the root 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 2.*

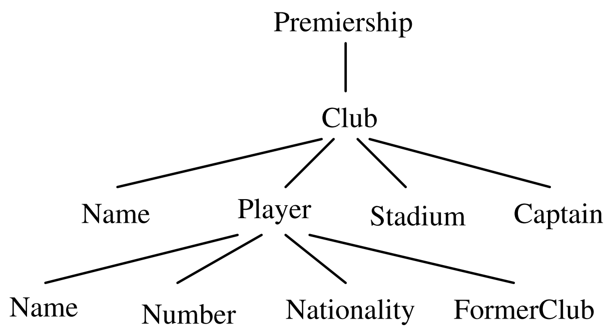

Figure 2. Displaying part of the FRO for the premiership object.

3.1.2 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 [1] and the path expressions expressible in it are called *general path expressions*. In our example we only use "?" that denotes an optional label and "%" that matches any number of characters.

Example 3.3 *Consider the following pattern* gpe $=$ Club(.Player)?.(Na%) *and the premiership object in Figure 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* $continuation_{premiership}(\epsilon) = \{$Club$\}$.
- *The label* Club *matches the head of* gpe, *the tail of* gpe *is* (Player.)?(Na%).
- *Then we find* $continuation_{premiership}$(Club) $=$ $\{$Name, Player, Captain, 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%.
- *We find* $continuation_{premiership}$(Club.Player) $=$ $\{$Name, Nationality, Number, 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.

3.1.3 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.

3.2. Degree-k representative objects

We obtain the following definition by replacing the $continuation$ function in Definition 3.1 by the $continuation^k$ function.

Definition 3.4 *Let* o *be an object and* $k \geq 1$. *Then* $continuation_o^k(pe) = continuation^k(o, pe)$, *where* 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 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 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 have the exact value of $continuation_o(pe)$.
- If $n = k$ we can find $continuation_o^k(pe)$ and by Lemma 2.9 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 also 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 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 where k is the degree of the k-RO 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.
- Path queries: The procedure described in the FRO case remains the same. When we compute continuation 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 one 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.

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 [1, 5]. 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 4.1 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 and $\perp$ if any of those objects is atomic.

Algorithm 4.1 *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 3 computes $continuation_o(pe)$.*

Input: o and $pe = l_1.l_2 \cdots l_n, n \geq 0$
Output: $continuation_o(pe)$

```
Let object set S = {o}
For i = 1..n
    Let object set T = {}
    For each object s ∈ S
        For each identifier id, such that ⟨l_i, id⟩ ∈ value(s)
            Add object(id) to T
    If T is empty then
        Return {}
    Else
        S = T
Endfor
Let label set C = {}
For each object s ∈ S
    If s is atomic then
        Add ⊥ to C
    Else
        For each object reference ⟨l, id⟩ ∈ value(s)
            Add label l to C
Endfor
Return C
```

Figure 3. Algorithm for computing $continuation_o(pe)$ from o.

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

Definition 4.2 *Let o_1 and o_2 be objects in OEM. Then o_1, along with Algorithm 4.1, is a* full representative object in OEM *for o_2 if for any simple path expression pe we have $continuation_{o_1}(pe) = continuation_{o_2}(pe)$.*

From Definition 4.2 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 4.3 *Formally, when we talk about FROs in OEM we always have to include Algorithm 4.1 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 4.1 and refer to the object part as the full representative object.*

4.1. Minimal FROs

Form Definition 4.2 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 4.1 to compute the $continuation$ function fastest. Each iteration of the first part Algorithm 4.1 takes time proportional to the size of S. Thus, the FRO for which the size of S at each iteration is smallest allows the fastest computation. The next definition describes a particular kind of FROs (in OEM) for which S always contains at most one complex object and at most one atomic object.

Definition 4.4 *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 the assertion that at each iteration of the first part (breadth-first exploration) of Algorithm 4.1 for R_o S contains at most one complex and one atomic object for any simple path expression. Before the first iteration the size of S is 1. Thus, for a simple path expression of length 0 (ϵ) the assertion holds since the first part of Algorithm 4.1 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 k-th iteration S contains more than one atomic objects or more than one complex objects. 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 S after the i-th iteration be o_i, for $i = 1..k-1$. At the k-th iteration S 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 $l_1.l_2 \cdots l_k$. This contradicts Definition 4.4 and thus the assertion holds in all cases.

We will use the assertion proved above to calculate the running time of Algorithm 4.1 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 4.1 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 4.1 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. 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 and minimization of this NFA that results in a deterministic finite automaton (DFA), and construction of a minimal FRO from this DFA. We also prove the correctness of this method.

5.1. Finite automata

Finite automata are used in many areas of computer science and are studied extensively[3]. A finite automaton $(Q, \Sigma, \delta, q_0, F)$ consists of a finite number 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.

5.2. 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 formally show 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 $\forall a \in \mathcal{A}$
- $q_0 = state(o)$
- $F = Q$

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 [3]. 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 [2].

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 $\delta(q, l) = r\}$.
- 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 5.1 *As an illustration of the method described in this section Figure 4 shows the minimal FRO in OEM for the premiership object in Figure 1. Note that there are two link labeled "Name" coming from the same "Club" object. This does not contradict Definition 4.4 because one of the "Name" subobjects is atomic and the other one is complex.*

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.

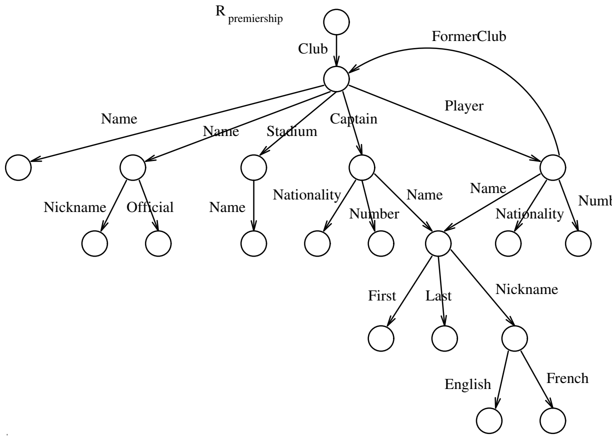

Figure 4. The minimal FRO for the premiership object.

Let o be an object, N_o the NFA constructed from o as described in Section 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 5.4. 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 $p = o, l_1, o_1 \cdots o_n$. 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 $i = 2..n$.

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.

6. Constructing a 1-representative object

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 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 6 shows the 1-RO for the OEM object in Figure 5. 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.

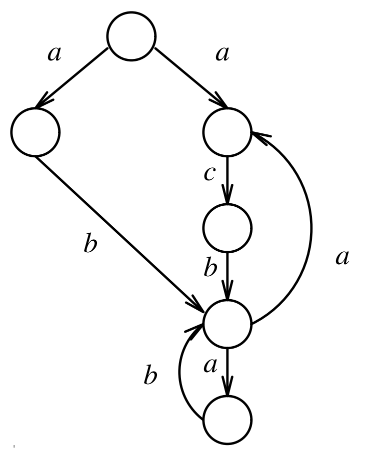

Figure 5. An example object.

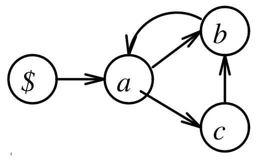

Figure 6. The 1-RO for the example object.

6.1. 1-representative object 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 on incoming links. Thus we must must examine all objects that have links to o_i. Our approach is to remember all $(identifier(o_i), l_{i+1}, identifier(o_{i+1}))$ triples (the id table) and join it with itself on $identifer(o_{i+1}) = identifer(o_i)$ to produce (l_{i+1}, l_{i+2}) pairs.

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

The result table, the label table, is then indexed by l_1 so that lookups are efficient. Duplicate label pairs are discarded.

6.2. Computing 1-continuations

Suppose that we wish to find the continuation of a simple path expression consisting of a single label l. Then we look for 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.

7. A graph-based approach to constructing k-ROs

Let us be given an object o in OEM. Let P be the set of simple path expressions, having length up to $k+1$, that have instance data paths within o. Specifically, we shall think of the set P as having strings of length $k+1$. To represent simple path expressions of length less than $k+1$, we pad them with the special label \$ at the beginning. We shall also refer to the elements of P as $k+1$-paths of o.

Evidently, the set of strings P is a suitable representation of the k-RO. It is also not hard to compute P; it requires a generalization of the technique discussed in Section 6 for the 1-RO. However, P is not a very compact representation of the k-RO. Thus, we shall show how to compact the set into a graph, from which k-continuations can be read efficiently.

7.1. Converting sets of strings into a compact graph

Our idea is to compact the set of simple path expressions into a graph such that all the paths of length $k+1$ in the graph have label sequences that appear within the given object, and conversely. One can then compute the k-continuation by searching the graph. An index on the nodes that directs us to all the nodes of the graph bearing a given label will make this search quite efficient.

A suitable graph may be constructed by listing all the $k+1$-paths of an object and partitioning the positions of those paths into clusters of mergeable positions. The nodes of the graph will each represent a cluster of positions. For positions to be mergable, they must surely have the same label, since they will be represented by a single node of the graph.

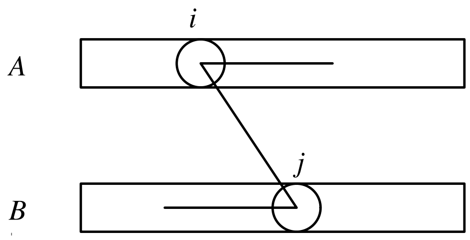

Figure 7. A cross-over string: requirement on positions that are mergeable.

However, there is also another condition they must satisfy. Suppose that we wish to merge position i of a $k+1$-path $A = a_1.a_2 \cdots a_{k+1}$ with position j of the $k+1$-path $B = b_1.b_2 \cdots b_{k+1}$. Then any $k+1$-path that we construct by starting with $b_m.b_{m+1} \cdots b_j$ and continuing with $a_{i+1}.a_{i+2} \cdots a_{k-j+i+m}$ must also be a $k+1$-path of the given object. Figure 7 suggests the $k+1$-path that must also appear in the given object. We call this $k+1$-path a *cross-over string*.

Example 7.1 *Figure 5 shows a graph representation of an object o in OEM. Let $k = 2$, so paths of length three are considered. The eight sequences of 3 labels (counting the special label \$) that appear in Figure 5 are shown in Figure 8.*

Consider paths (6) and (7), and suppose we wish to merge the last position of (6) with the first position of (7). Both hold label b, so it is possible that the merger will succeed. We need to consider a string of 3 labels, beginning in (6), reaching the b at the end, treating that b as if it were the first position of (7), and continuing in (7) until a total of three

1) $\$\$a$
2) $\$ab$
3) $\$ac$
4) acb
5) aba
6) bab
7) bac
8) cba

Figure 8. Length-3 paths of the example object.

positions are visited. That sequence of three positions must also be on the list of Figure 8.

The only way the new sequence could not be on the list is if we take the middle positions of (6) and (7), along with the merged b. This string is aba, and it is string (5) in Figure 8. In fact, in this example, every position bearing the same letter can be merged. It is also possible to merge positions 2 of string (1) with positions 1 of strings (2) and (3). The only case in which positions bearing the same label cannot be merged is that position 1 of string (1) cannot be merged with the other positions holding \$. If we make one node of the graph for each cluster of mergeable positions, we get the graph shown in Figure 9.

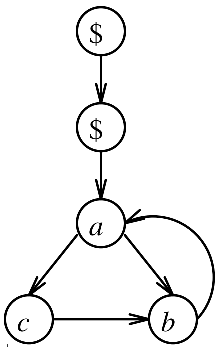

Figure 9. Graph constructed for the example object.

Observe that things are not always as simple as suggested by Example 7.1. For instance, suppose we remove the lowest object with label b within the object o of Figure 5, and call the resulting object o'. The set of 3-paths for o' is the same as for o, except it is missing the string (6): bab. That would prevent merging many pairs of positions. For instance, we could not merge the first and third positions of string (5), which is aba, because that would require that string bab were also present in the object o'. In that case, the best we could do would be to use a copy of o' itself, with two nodes labeled \$ above the top a in Figure 5, and with the bottom b deleted, of course.

There are several simplifications that can be made to our test for whether two positions are mergeable.

- First positions with the same label are always mergeable. The reason is that each of the strings induced by the cross-over process suggested in Figure 7 must be one of the two strings involved in the merger.
- Similarly, last positions with the same label are surely mergeable.
- If $j < i$, then there is no room for a cross-over string as in Figure 7, so unless $i = j$ we only have to consider cross-over strings that begin in one of the two strings whose positions we are considering merging (the one with the further right of the positions being merged).
- When we merge strings, a new cross-over string could only be obtained if we use something from each string, other than the merged positions. In terms of Figure 7, the cross-over string must begin before position j of B, and it must end after position i of A. Thus, $m < j$ may be assumed.
- Thus, in the special case $k = 2$, the only cross-over strings that can prevent a merger are those with one position before the merged position of one string and one position following the merged position of the other.

7.2. Computing k-continuations

The data structure used to represent the graph influences how fast we can compute k-continuations. For maximum efficiency, we need to have a *main* index that maps labels to the set of nodes with that label. We also need to have for each node an index mapping labels to the successors of that node having that label.

Assume these structures are available and can retrieve the desired set of nodes in time proportional to the size of the set. Suppose we wish to know the continuations of path $l_1.l_2.\cdots.l_k$. Then we use the main index to find the set of nodes labeled l_1. For each of these nodes, we use the index for that node to find the successors labeled l_2, and so on.

While this search could be exponential, we only need to find successors of each node at most once for each position in the string. If we keep track in a table of those pairs (i, n) such that we found for node n (whose label must be l_i) the successors of n with label l_{i+1}, then the total amount of work we do is at most kN^2, where N is the number of nodes of the graph.

8. A finite automaton-based approach to constructing k-ROs

In Section 7, we presented a graph-based k-representative object. The graph encodes the set P of simple path expressions of length up to $k+1$ that have instance data paths within the object being represented. In this section, we present a construction for k-representative objects based on finite automata. We treat simple path expressions as strings over the alphabet of OEM labels. The language represented by the set P is then encoded using an automaton that accepts the strings in P.

8.1. Constructing an automaton representing an object in OEM

As in Section 7, 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 [3], we construct a finite automaton A that recognizes the language P. (We assume that the automaton A is minimized using the subset construction method [3].)

Consider the example in Section 7, involving the object in Figure 5. Figure 8 shows the 3-paths of 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 10. The initial state is marked with a short arrow, and the accepting states are circled.

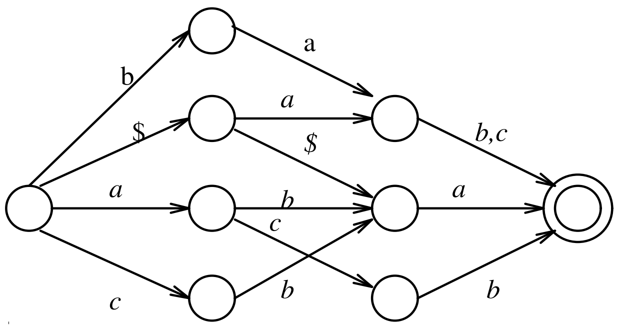

Figure 10. Finite-automaton-based 2-RO for the example object.

8.2. 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 $a_1.a_2 \ldots a_k$. We start in the initial state of the automaton and follow the transition with label a_i for $i = 1..k$ 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 $O(k+c)$ if we use an associative array for the label-index. Thus, the time required to find the k-continuation is bounded by $O(klogl + c)$.

8.3. Comparison

For the object of the example of Figure 5, the finite automaton-based 2-representative object is more complicated than the graph-based 2-representative object in Figure 9. However, for other objects, the finite automaton-based representative object is simpler than the graph-based one.

9. Conclusions

In this paper we have introduced the representative object concept that provides a concise representation of the inherent schema of a semistructured hierarchical data source. We make the case that representative objects are very useful for semistructured data and show some of their primary uses. We also described an implementation of 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 has 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.

9.1. Future work

We are investigating the following topics.

- Storing information about the typical object in the FROs. The FROs provide information about the overall schema of the data but do not provide any information about the instance objects. For example, an FRO can tell you that a link labeled "Book" can only be followed by links labeled "Author", "Title", and "Publisher" but cannot tell you if every link labeled "Book" is followed by a link labeled "Title".
- Graph-based construction algorithms for k-ROs. We are looking at more complicated conditions that can help the graph-based approach to constructing k-ROs.
- Updating minimal FROs in OEM. When the object changes its minimal FRO in OEM also must change. Since the construction of a minimal FRO from scratch is expensive we need to have a way of updating the old minimal FRO accordingly.

References

[1] 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`.

[2] 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.

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

[4] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, Mar. 1995.

[5] D. Quass and et. al. Lore: A lightweight object repository for semistructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 549, Montreal, Canada, June 1996.

[6] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, Rockville, Maryland, 1989.