Querying Composite Objects in Semistructured Data

Keishi Tajima
Department of Computer and Systems Engineering,
Kobe University, Japan
tajima@db.cs.kobe-u.ac.jp

Abstract

In this paper, we propose an entity-based style of queries for semistructured data. First, we partition a semistructured data into subgraphs corresponding to real-world entities, in other words, into composite objects. To detect composite objects in semistructured data, we use the exclusiveness of references. If a reference is exclusive, then we regard is as a composite link. Then, we develop a query language for entity-based queries. That language supports path expressions, in which we can use edge expressions that match only with composite links or non-composite links. By using these expressions in combination with wild cards, we can specify queries of a form like "retrieve all entities including these data items," which we call entity-based style queries. We show examples demonstrating how this style of query is useful especially when one does not have enough knowledge on the schema in advance.

Keywords semistructured data, composite object, structure discovery, path expressions

1 Introduction

Semistructured data is schema-less, self-describing data [1, 4]. In many researches, data models for semistructured data are proposed [12, 5, 11, 2, 8]. Those data models slightly differ from each other, but basically they all represent semistructured data in a form of edge-labeled directed graph. Nodes in a graph correspond to data objects, edges from a node correspond to references from that object to other objects, and labels on those edges correspond to attribute names describing the meaning of those references.

For example, Figure 1 shows a graph representing a semistructured data. This data represents a part of a movie database. The root node at the top of the figure works as the entry point of the whole database, and it has references to all entries of movies and actors. Each movie entry has two attributes title and cast, and each actor entry has two attributes name and appear. Those attributes are represented by the edges outgoing from the root node of each entry. Note that we assume that one node may have multiple outgoing edges with the same label because most data models proposed for semistructured data allow it to improve flexibility. In some cases, an attribute refers to a node representing another entry. For example, the attribute cast of the entry of the movie "Godzilla" refers to the entry of the actor "Matthew Broderick." In some cases, an attribute refers to a node representing a primitive value. For example, the attribute title of the entry of the movie "Godzilla" refers to a node representing a string value "Godzilla." In addition, in some cases, attribute values are further organized into a hierarchy. For example, the attribute name of the entry of the actor "Matthew Broderick" is further organized into a hierarchy consisting of a root node, two edges with labels first and last, and nodes representing the values of those two attributes.

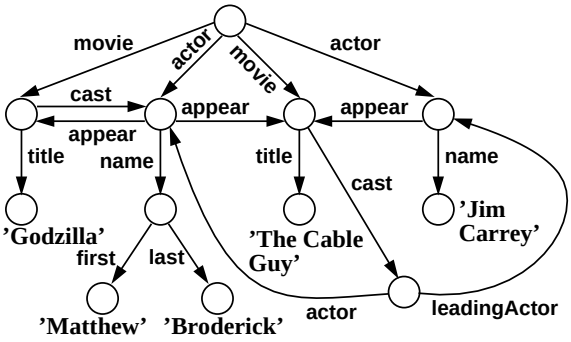

Figure 1: An Example of Semistructured Data

As shown in the example above, semistructured data is represented by a big continuous graph consisting of uniform nodes and edges. In most cases, however, as also shown in the example above, that big graph consists of subgraphs representing various kind of real-world entities and references between them. For example, the example database graph above consists of four entities, i.e. the movie "Godzilla," the actor "Matthew Broderick," the

To appear in Proceedings of the 5th International Conference on Foundations of Data Organization (FODO'98), Kobe, Japan, Nov. 1998.

movie "The Cable Guy," and the actor "Jim Carrey," and references between them. As shown in Figure 1, each entity in semistructured data is represented as a single-rooted tree.

Because of the lack of the rigid schema, subtrees representing the same kind of entities may have different structures. For example, two actor entries in the above example are referring to different structure through name attribute. In the same way, two movie entries are referring to different structure through cast attribute. This structural heterogeneity makes set-oriented operations on those semantically homogeneous sets of entities difficult.

To solve this difficulty, many query languages for semistructured data support path expressions including wild cards. For example, consider a query "list the titles of all movies that feature an actor named Carrey." In those query languages, this kind of queries are expressed by using wild cards, such as in the following way[1] :

```
select l
where movie⇒ t, t ⇒title⇒ l, t ⇒cast⇒[_⇒]*n,
      substring(n, 'Carrey')
```

In this query, where clause describes path patterns that should be matched with. "_" is an anonymous variable that matches with any label, and * means the repeat of any number of times including zero. Thus, the expression [_⇒]* as a whole is a wild card that matches with any path of arbitrary length. This wild card is used because the structures under cast attribute is heterogeneous, and the users are uncertain of those structures. The path patterns in the where clause of this query matches with subtrees in the database that consists of an edge with a label movie emanating from the root of the database, and two paths beneath it, one of which is starting with the label title followed by some value l, and the other of which is a path of arbitrary length starting with the label cast and ending with some value n which include "Carrey" as its substring. For each matched subtree in the database, the qeury returns the value l.

In some cases, however, such wild cards that match with paths of arbitrary length happen to match with unexpected paths. For example, when we apply the query above to the database shown in Figure 1, the pattern cast⇒[_⇒]*n matches with the path starting cast edge from the entry of the movie "Godzilla" to the entry of the actor "Broderick," going through appear edge to the movie "The Cable Guy", going through cast and leadingActor edges to the actor "Jim Carrey", and ending with the label "Jim Carrey" in that entry. Therefore, the result of the query above includes the movie title "Godzilla" although Jim Carrey does not appear in the movie "Godzilla."

One approach to avoid that kind of unexpected matching is to use more complicated regular expressions so that they eliminate those unexpected matching. For example, the query above can be rewritten as below:

```
select l
where movie⇒ t,
      t ⇒title⇒ l, t ⇒cast⇒[^appear⇒]*n,
      substring(n, 'Carrey')
```

In the query above, [^appear]* is a regular expression specifying any paths not including the label appear. The result of this new query does not include the unwanted title "Godzilla."

Specifying appropriate regular expressions that match only with really needed paths is, however, quite difficult task. It is because paths of arbitrary length may reach to everywhere in the database graph, and it is difficult if not impossible to anticipate all "unexpected" cases. Our insistence is that wild cards that match with paths of arbitrary length are essentially dangerous. In some cases, we certainly need such wild cards, but in most cases, more restricted use of wild cards must be more appropriate. For example, the intention of the expression cast⇒[_⇒]*n in the first query is to skip some heterogeneous structure within the cast attribute of movie entries, and is not to reach everywhere in the database through other actor entries and movie entries. Therefore, in this case, we should restrict that expression to match only with paths within a substructure in a movie entry.

From the observation above, in this paper, we propose a query language for semistructured data that supports constructs to controll the range of wild cards. Our basic strategy is to detect each entity in the database, and to specify whether each wild card in a query is allowed to match with paths extending over multiple entity or it matches only with paths within one entity.

As mentioned before, by the term entity, we mean logical data unit corresponding to each complete and independent real-world entity. Entities in semistructured databases are represented as rooted connected trees. One similar conecpt that has been proposed in the past database researches is *composite objects* [7] in the object-oriented database researches. A composite object is a hierarchy consisting of objects that are strongly connected through is-part-of relationships. Although a composite object consists of multiple part objects, it is regarded as one independent logical data unit as a whole. The underlying concept of composite objects is that objects have two kinds of references: those representing exclusive possesion of their part objects, and those representing various non-exclusive rela-

[1] The syntax used in this example is ambiguous. We will define more rigid syntax later.

tions between objects. The former type of references are sometime called *composite links*, and the latter are sometime called *non-composite links*. Composite objects are connected subgraphs consisting of only composite links.

Then, the next problem is how to detect such entities, i.e. composite objects, in semistructured data. In this research, we use the exclusiveness of references. As suggested in the research of composite objects, we consider that entities are subtrees consisting of only exclusive references. Based on this assumption, we statically divide a database graph into entities by examining each reference is exclusive or not. This approach, of course, must not always work perfectly. Queries on semistructured data are, however, essentially imperfect in most cases because the user does not know all the data structure appearing in the given semistructured databases. We believe our method at least improves the correctness of queries on semistructured data.

The rest of the paper is organized as follows. In the next section, we briefly review related work. In Section 3, we explain a datamodel and a query language that we use as the base of our development. Then, in Secion 4, we discuss the composite object detection, and introduce new constructs to controll the range of wild cards. Finally, Section 5 is the conclusion.

2 Related Work

There is a couple of researches on finding structures in semistructured data. In [10, 9], they propose a method to infer the approximate typing of every nodes in semistructured data. In their researches, for each node, they infer its type in accordance with the type of the nodes to which that object referrs and the labels of those references, and in the same way, the type of the nodes referring to it and the labels of those references. Their goal is to find types in data, i.e. classes of nodes with similar structure. On the other hand, what we want to do in this research is to partition a database graph into subgraphs representing independent logical data units, i.e. composite objects corresponding to real-world entities. Therefore, these two researches and our research focus on the extraction of different kind of schema information out of semistructured data. The information on types of each node, however, can help the detection of composite objects. We will explain how the type information can help the detection of composite objects later in this paper.

[13] is also focusing on finding structure in semistructured data. In that paper, they try to identify nodes that should be regarded as instances of classes. Their approach is to regard every labels followed by primitive values as attribute names, and to regard every nodes followed by attribute names as instances of classes. By that method, the node referring to the nodes "Matthew" and "Broderick" in Figure 1 is regarded as an instance of a class because that node is regarded as referring to those primitive values through two attributes named first and last. What we want to find in this research is, however, not nodes that may be instances of classes but nodes that seem to be the roots of composite objects. In our sense, the node referring to "Matthew" and "Broderick" is exclusively posessed by an actor entry, and just a substructure of a composite object representing an actor entity.

There is also a couple of researches on finding structures in hypertext data. In [3], they propose a framework to provide users with an abstracted overview of hypertext by aggregating strongly related nodes. In their approach, they regard a set of strongly connected nodes, such as nodes in biconnected subgraphs, as composing a logical data unit. It is because the data unit they want to find is sets of strongly related nodes, and in hypertext data, strongly related nodes tend to be strongly connected by mutual references or cyclyc links. On the other hand, the data unit we want to find is composite objects. Although composite objects are, in some sense, also strongly related nodes, nodes in one composite object in semistructured data do not seem to be always strongly connected in the sense of biconnectivity and so on. Exclusiveness of references seems more appropriate for our purpose. In this way, we use a compltely different method to detect a different kind of structure in the data.

Another research on finding structure in hypertext data is our previous research [14]. In that research, we develop a method to parition a hypertext data into independent subtopic structures. Because the data structure we want to find in that research is subtopic structures in hypertext data, we use similarity of the contents of nodes in order to detect them. Therefore, the kind of structures to find and the method to detect those strucutres are completely different from this research.

[6] is discussing the extraction of information on individual real-world entities, such as persons or companies, out of semistructured data. Their purpose is to automatically gather information on each person or company out of heterogeneous multiple resources. Therefore, their purpose is different from our purpose in this research. One basic assumption is, however, common to both researhces. Both they and we consider that the typical queries issued on semistructured data are entiti-based. For example, the query "list the titles of all movies featuring an actor named Carrey" discussed above is certainly based on the concept of "movie" entities and "actor" entities.

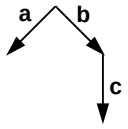

Figure 2: An Example of A Tree

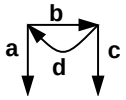

Figure 3: An Example of A Graph

3 Basic Model

In this research, we use the eadge-labeld tree datamodel and UnQL, which are a data model and its query language for semistructured data proposed in [5], as the base model for the development. In this section, we breifly explain them.

In this datamodel, semistructured data is represented in a form of a directed tree with labels on its edges. The syntax for the construction of edge-labeled trees is defined as below[2] :

$$\begin{aligned} tree &::= \{label \Rightarrow tree, \ldots, label \Rightarrow tree\} \mid \\ &\quad\ tree{-}marker \\ label &::= int \mid string \mid \ldots \mid symbol \end{aligned}$$

tree is defined as either a set of pairs of a label and a tree ($\{label \Rightarrow tree, \ldots, label \Rightarrow tree\}$ in the definition above) or a tree-marker, which is explained later. *label* can be any values of supported primitive types, such as integer, string, and symbol. By using this syntax, a tree in Figure 2, for example, is described as below:

{a⇒{}, b⇒{c⇒{}}}

Note that leaves of a tree are represented by empty trees, i.e. empty sets. In this model, primitive values are represented by labels of edges followed by no edges beneath them. From now on, we use a syntax sugar, and simply write a for a⇒{}. We also omit braces ({ }) for singleton sets. For example, we write as below instead of the description above:

{a, b⇒c}

This model can also describe cyclic structure using tree-markers and one more construct, letrec. For example, the rooted graph shown in Figure 3 is described as below:

letrec X={a, b⇒Y}, Y={c, d⇒X} in X end

[2] This definition is slightly different from the one in [5].

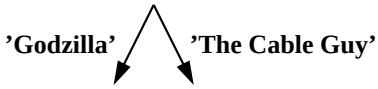

Figure 5: The Result of the Example Query

This definition define two trees X and Y which mutually refer to one another, and define the root node of the tree X, i.e. a node with outgoing edges a and b in Figure 3, as the root.

Figure 4 is an example database represented in the edge-labeled tree model. This database is a movie database, and it contains two kinds of entities, movies and actors. Movie entries have attributes title, year, country, and cast, and actor entries have name, sex, and appear. This data can be described by the syntax above as below:

```
letrec
  X =  {movie⇒M1, movie⇒M2,
        actor⇒A1, actor⇒A2}
  M1 = {title⇒'Godzilla', year⇒1998,
        country⇒'U.S.A.', cast⇒A2},
  M2 = {title⇒'The Cable Guy',
        year⇒1996, country⇒'U.S.A.',
        cast⇒{leadingActor⇒A1,
              actor⇒A2}},
  A1 = {name⇒'Jim Carrey', sex⇒male,
        appear⇒M2},
  A2 = {name⇒{first⇒'Matthew',
              last⇒Broderick'},
        sex⇒male,
        appear⇒M1, appear⇒M2}
in X end
```

Next, we explain the query language by showing intuitive examples rather than by formally defining it. On the database above, we can issue queries like below:

```
select t
where movie⇒title⇒ \t ← DB
```

The construct ... ← DB in where clause produces the paths emanating from the root of the database graph bound to the name DB. Here, we assume the database shown above is bound to the name DB. Then, each path is compared with the pattern at the left hand of ←. For each path that matches with the pattern, the variables newly introduced in the pattern are bound, and select clause is evaluated. Newly introduced variables are marked with \. The result of the query is the union of the result of all the evaluation of select clause. In the query above, t is bound to a tree under an edge labeled title, and the result of each evaluation of select clause is a tree, i.e. a set of edges, and therefore, the result of the query is the union of trees, which

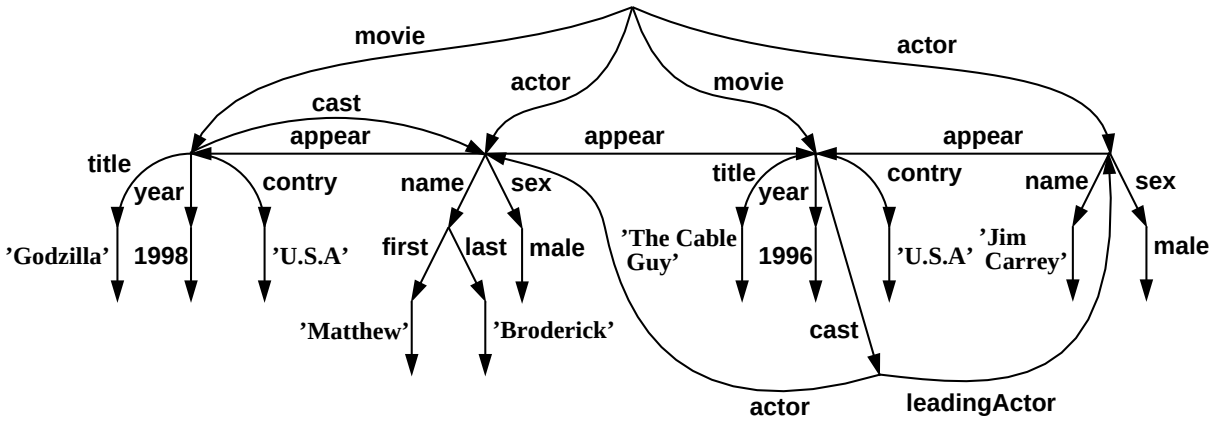

Figure 4: An Example Movie Database

is also a tree. The result of the query above is a tree shown in Figure 5.

If we want to get not the tree in Figure 5 but a plain set of title strings, the query below does for it:

select l
where movie⇒title⇒ $\backslash l \Rightarrow \{\} \leftarrow$ DB

While the variable t at the end of the path expression in the previous query matches with a whole subtree under the path, the variable l in this query matches with the label, which must be a title of a movie, because it is followed by $\Rightarrow \{\}$. The result of this query is {'Godzilla', 'The Cable Guy'}.

The query "list the titles of all movies featuring an actor named Carrey," discussed in the introduction, is described as below[3] :

select l_1
where movie⇒ {title⇒ $\backslash l_1 \Rightarrow \{\}$,
cast⇒[^appear⇒]*$\backslash l_2 \Rightarrow \{\}\}$
$\leftarrow$ DB,
substring(l_2, 'Carrey')

As explained before, [^appear⇒]* is a regular expression that matches with any paths not including a label appear. Note that $\Rightarrow \{\}$ following $\backslash l_1$ and $\backslash l_2$ are needed so that l_1 and l_2 are bound not to a tree but to a label as explained above.

4 New Constructs to Control Wild Cards

Now in this section, we introduce new constructs to control the range of wild cards. First, we explain the method to detect composite objects in semistructured data.

[3] The syntax for regular expressions used here is slightly different from the one in [5]

4.1 Composite Object Detection

As mentioned in the introduction, we consider connected subgraphs consisting of only exclusive references to be composite objects.

Definition 1 *A node is a root of a composite object iff there are multiple nodes referring to it. Otherwise, that node is possessed by its only parent object. The root node of the database is the exception, and it is always the root of a composite object. Then, a composite object is a subgraph $\langle V, E\rangle$ that can be extracted by the following steps:*

1. *let $\langle V, E\rangle$ be $\langle\{n\}, \emptyset\rangle$ where n is a node that is determined as a root of a composite object, and*
2. *repeatedly add to V nodes possessed by any node in V, and to E edges between nodes in V.*

∎

This fairly simple approach using the exclusiveness of references works well with the example database above. By this method, the example database is successfully partitioned into 5 composite objects: 2 movies, 2 actors, and the root node.

In some cases, however, this simple method may not be perfect. For example, suppose that the root node in the example data has only links to movie entries, and actor entries can be accessed only through related movie entries. Then, if some actor appears in only one movie stored in this database, that actor is regarded as exclusively possessed by that movie and as a part of the composite object representing that movie. In the example data above, the actor "Jim Carrey" is related to only one movie, and therefore, if not for actor edges from the root node, the entry of "Jim Carrey" would be regarded as a part of larger composite objects.

One approach to solve this problem is to use typing techniques proposed in [9]. Before determining composite objects, we first detect approximated types of all nodes by their techniques, and change the first sentence of the definition of composite objects above as follows:

Definition 2 *A node n is a root of a composite object iff there is a node n' of the same type as n and there are multiple nodes referring to n'. ...* ∎

If we can detect that the nodes rooting each actor entry have the same approximated type, i.e. if we can detect that those nodes share similar structure, then we can determine each actor entry as a composite object as long as there is at least one actor appearing more than tow movies stored in the database.

This approach depends on the accuracy of the approximated typing. Approximated typing has a parameter on the allowable heterogeneity in a single type. If we allow no heterogeneity in one type, all nodes of the same type have exactly the same structure, but in worst case, we have as many types as nodes. On the other hand, if we allow high heterogeneity, the number of types is decreased, but the sets of nodes of the same types are more heterogeneous. In the detection of composite objects, if we use a result of approximated typing with lower allowance, the precision rate of composite object detection will be higher but the recall rate will be lower. On the contrary, if we use a result of approximated typing with higher allowance, the recall rate will be higher but the precision rate will be lower.

4.2 Constructs for Restricted Edge Matching

Now we introduce a new construct $\langle label \Rightarrow \rangle^\circ$ to restrict the matching of edges. The pattern $\langle label \Rightarrow \rangle^\circ$ matches only with edges that has label *label*, and in addition, that does not across a boarder of a composite object. In other words, $\langle label \Rightarrow \rangle^\circ$ matches only with composite links. This construct is, for example, used in the following way:

```
select l
where movie⇒{title⇒'Godzilla',
                [⟨_⇒⟩°]*year⇒ \l ⇒ {}} ← DB
```

This query is to retrieve the value of year attribute of the movie "Godzilla." $\langle \_\Rightarrow\rangle^\circ$ matches with composite links with any labels. Therefore, the pattern movie⇒[$\langle$_⇒$\rangle^\circ$]*year⇒ matches with paths starting with the label movie, going through arbitrary number of composite links, and ending at an edge with the label year. By restricting the intermediate edges to be within a singe composite object, this query tries to find a year edge at some unknown place but inside the movie entry for "Godzilla." (In the example database above, all year edges are immediately under movie edges, but the users may be uncertain of it, or there may actually be movie entries in which year edges exist at deeper place because the data is semistructured data.) When this query is applied to the example database above, it correctly returns a set {1998}.

For the symmetricity, we also introduce a construct $\langle label \Rightarrow\rangle^\times$, which explicitly requires non-composite links. We actually sometime need such a construct. For example, consider the query below:

```
select l where
  movie⇒
  {title⇒'Godzilla',
   [⟨_⇒⟩°]*⟨_⇒⟩×[⟨_⇒⟩°]*name⇒ \l ⇒ {}}
  ← DB
```

This query tries to retreive names of people related to the movie titled "Godzilla." $\langle \_\Rightarrow\rangle^\times$ is used because there may be name edges inside the movie entry itself.

By using these new constructs, the query "list the titles of all movies featuring an actor named Carrey," discussed in the introduction is now can be specified as follows:

```
select l1 where
  movie⇒
  {title⇒ \l1 ⇒ {},
   [⟨cast⇒⟩×[⟨_⇒⟩°]*  |
    ⟨cast⇒⟩°[⟨_⇒⟩°]*⟨_⇒⟩×[⟨_⇒⟩°]*]\l2 ⇒ {}}
   ← DB,
  substring(l2, 'Carrey')
```

[... | ...] is a disjunction pattern that specifies either side of | should be matched. We define the semantics of a query including a disjunction pattern by the union of two queries corresponding to two alternative patterns. Therefore, each alternative pattern has its own scopf of variables, and in select clause or at other places in where clause, variables that are introduced in the both alternative patters can be used, while variables that are introduced in only one of alternatives cannot be used. The pattern ⟨cast⇒⟩×[⟨_⇒⟩°]* at the left sied of | matches with the paths starting with a non-composite link labeled cast, and going through arbitrary number of composite links. On the other hand, the pattern ⟨cast⇒⟩°[⟨_⇒⟩°]*⟨_⇒⟩×[⟨_⇒⟩°]* at the right side of | matches with paths starting with a composite cast link, going through arbitrary number of edges including only one non-composite link. Thus, the whole expression [... | ...] matches with paths starting with a cast edge, having arbitrary length, and including only one non-composite link somewhere in it. We need two alternative patterns because it is not certain whether a cast link is a composite link or a non-composite link.

By these restrictions in the path expressions, when this query is issued on the example database in Figure 4, it correctly returns {'The Cable Guy'}. It does not include "Godzilla" in the result.

As shown in the examples above, with those new costructs we can use wild cards in a more controlled way. The examples above, however, also shows that the pattern specification would be quite complicated. To make query specifications simpler and easier to write, we introduce one macro expression. The typical way of the use of those new constructs is the one in the last query: "this pattern matches only with paths including this number of composite links and this number of non-composite links." Therefore, we introduce a macro that directly specifies that kind of condition. A macro expression $\langle \textit{path pattern} \rangle^{\circ n_1}_{\times n_2}$ means a path pattern that matches only with paths including n_1 composite links and n_2 non-composite links. We can omit either n_1 or n_2, in which case the path may include any number of links of that type. We do not formally define the translation of this macro expression in this paper because it is very complicated, but it is possible to define an automatic translation. By using this macro, the previous query can be rewritten as follows:

```
select l1
where movie⇒ {title⇒ \l1 ⇒ {},
                ⟨cast⇒[_⇒]*⟩°×1\l2 ⇒ {}}
        ← DB,
      substring(l2, 'Carrey')
```

This specification is translated into the previous query before execution and returns the same result.

In the same way, the query "retreive names of people related to the movie "Godzilla" shown above can be rewritten as follows:

```
select l
where movie⇒{title⇒'Godzilla'⇒ {},
                ⟨[_⇒]*⟩°×1name⇒ \l ⇒ {}}
        ← DB
```

4.3 Entity-Based Query in Semistrucutred Data

As illustrated in the previous subsection, our constructs that explicitly require compsite links or non-composite links are useful especially when they are used in combination with wild cards. They can be used to controll the range of wild cards, and it makes wild cards safer and more usable than usual wild cards that match with paths of arbitrary length. Being able to use wild cards more often implies we can specify successful queries with less knowledge on the schema. In this subsection, we demonstrate a style of query specifications that maximize the benefit of wild cards, which we call entity-based query style.

Although the example queries shown above are queries on semistructured data and does not assume the users' complete knowledge on the schema, they still assume some knowledge on it. For example, the first query retrieving the value of year attribute is assuming that a movie edge appears immediately under the root and a title edge appears immediately under movie. We, however, need not assume them at all if we take the full advantage of wild cards. That query can be rewritten as below:

```
select l
where [_⇒]*{[⟨_⇒⟩°]*'Godzilla',
            [⟨_⇒⟩°]*year⇒ \l ⇒ {}} ← DB
```

This query assumes only a very little knowledge on the schema. To see it, suppose we remove the restriction on link types in the query above. If you remove all those resticrtions, this query almost does not make sense. That query will return all the labels appearing under the label year if there is at least one label 'Godzilla' in the database, or otherwise will return an empty set. With those restrictions by our new constructs, however, this query correctly returns the production year of "Godzilla."

In the same way, if we want to take full advantage of wild careds, the query "list the titles of all movies featuring an actor named Carrey," is rewritten as below:

```
select l1
where [_⇒]*{[⟨_⇒⟩°]*title⇒ \l1 ⇒ {},
            [⟨_⇒⟩°]*⟨cast⇒[_⇒]*⟩°×1\l2 ⇒ {}}
        ← DB,
      substring(l2, 'Carrey')
```

Part of these queries have a common style: they try to find entities including all the required items. This style of queries using as little knowledge on the schema as possible and focusing on the concept of entities seems a typical query style in many applications. We call this style of query *entity-based query style*, and in order to promote this style of query, we introduce a macro expression for it. Instead of writing $[\_\Rightarrow]^*\{[\langle\_\Rightarrow\rangle^\circ]^*t_1, \ldots, [\langle\_\Rightarrow\rangle^\circ]^*t_n\}$, we write $E\{t_1, \ldots, t_n\}$. By using these macros, two queries above are rewritten as follows:

```
select l
where E{'Godzilla', year⇒ \l ⇒ {}} ← DB
```

and

```
select l1
where E{title⇒ \l1 ⇒ {},
          ⟨cast⇒[_⇒]*⟩°×1\l2 ⇒ {}}
        ← DB,
      substring(l2, 'Carrey')
```

Although the second one is still a little complicated, the first one became greatly easier to read.

We show a couple of more examples demonstrating entity-based style queries. The query below returns the list of titles of all movies producted in U.S.A. in 1996:

select l
where $E\{\mathsf{title} \Rightarrow \backslash l \Rightarrow \{\}, \mathsf{1996} \Rightarrow \{\},$
$\mathsf{"U.S.A."} \Rightarrow \{\}\} \leftarrow \mathsf{DB}$

Although in the example database above, year and country are always immediately under movie, there may be entries or another database where they are not. The query above is assuming as little knowledge as possible, and therefore, very portable.

The query below returns all the labels of terminating edges within the entry of "Jim Carrey," which must be values of primitive attributes in that entry:

select l
where $E\{\backslash l \Rightarrow \{\}, \mathsf{"Jim\ Carrey"} \Rightarrow \{\}\} \leftarrow \mathsf{DB}$

When this query is applied to the example database above, it returns a set {"Jim Carrey", male}. This query can be used to list all the primitive values describing the actor "Jim Carrey." On the other hand, the query below is also interesting:

select l
where $E\{\backslash l \Rightarrow \backslash t, \mathsf{"Jim\ Carrey"} \Rightarrow \{\}\} \leftarrow \mathsf{DB},$
$t \neq \emptyset$

This query returns {name, sex}. This can be regarded as answering what kind of information are known about "Jim Carrey."

5 Conclusion

In this paper, we propose a query language for semistructured data. Our language supports path expressions including wlid cards, and in addition, edge expressions that match only with composite links or non-composite links can be used in path expressions. By using these expressions in combination with wild cards, we can use wild cards in a more resticted and safer way. For example, by using these expressions and wild cards, we can specify a path expression that matches paths of arbitrary length but not extending to other entities. We showed examples demonstrating how those controlled wild cards are useful, and finally propose a style of query specification that take full advantage of them, which we call entity-based query style. As a method to distinguish composite links and non-composite links in semistructured data, we propose a method that uses the exclusiveness of references.

Our method of entity detection depends on the way of data organization of the given data. In this research, we assumed heirarchical style of data organization of semistructured data, where databases are consisting of subtrees representing real-world entities, and references between them. In many applications, this assumption seems correct. In some applications, however, databases have completely different structures, for which our approach is not useful. An example of such structures is doubly-linked lists, which are often used in hypertext document data. In hypertext document data, it is often the case that a single document is divided into its subparts, such as sections, and those subparts are arranged in a form of doubly-linked lists. In such a situation, although those subparts can be regarded as composing a single logical data unit, i.e. one document, our method using exlusiveness of references would determine that each subpart is an independent composite object because they all have multiple incoming links, one from the previous section, and one from the next section.

Even in such a situation, however, if we can correctly detect logical data units in the data, our new constructs and entity-based style of queries must be useful. Therefore, one important future work is to develop other methods of composite object detection for those kinds of data structures.

In addition, even in the applications where databases are basically consist of subtrees corresponding to entities and references between them, our method of entity detection is not, of course, perfect. Because the usefulness of our new constructs for path expressions and entity-based query style is crucially depend on the accuracy of entity detection, we must also improve techniques of entity detection even for those kind of data in future work.

Acknowledgments

We would like to thank Katsumi Tanaka for his continuous support to our research. This research is partly supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research on Priority Area (A): "Advanced databases," area no. 275 (08244103). This research is also supported in part by "Research for the Future" Program of Japan Society for the Promotion of Science under the Project "Advanced Multimedia Contents Processing" (Project No. JSPS-RFTF97P00501).

References

[1] Serge Abiteboul. Querying semi-structured data. In *Proc. of ICDT*, volume 1186 of *LNCS*, pages 1–18. Springer-Verlag, Jan. 1997.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal of Digital Libraries*, 1(1):68–88, Apr. 1997.

[3] Rodrigo A. Botafogo and Ben Shneiderman. Identifying aggregates in hypertext structures. In *Proc. of Hypertext*, pages 63–74, Dec. 1991.

[4] Peter Buneman. Semistructured data. In *Proc. of ACM PODS*, pages 117–121, May 1997.

[5] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proc. of ACM SIGMOD*, pages 505–516, Jun. 1996.

[6] Scott Huffman and Catherine Baudin. Notes explorer: Entity-based retreival in shared, semi-structured information spaces. In *Proc. of ACM CIKM*, pages 99–106, Nov. 1996.

[7] Won Kim, Elisa Bertino, and Jorge F. Garza. Composite objects revisited. In *Proc. of ACM SIGMOD*, pages 337–347, Jun. 1989.

[8] Alberto O. Mendelzon and Tova Milo. Formal models of Web queries. In *Proc. of ACM PODS*, pages 134–143, May 1997.

[9] Svetlozar Nestorov, Serge Abiteboul, and Rajeev Motwani. Extracting schema from semistructured data. In *Proc. of ACM SIGMOD*, pages 295–306, Jun. 1998.

[10] Svetlozar Nestrov, Serge Abiteboul, and Rajeev Motwani. Inferring structure in semistructured data. In *Proc. of Workshop on Management of Semistructured Data (in Conjunction with PODS/SIGMOD)*, May 1997. http://www.research.att.com/~suciu/workshop-papers.html.

[11] Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Object fusion in mediator systems. In *Proc. of VLDB*, pages 413–424, Sep. 1996.

[12] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *Proc. of IEEE ICDE*, pages 251–260, 1995.

[13] Dong-Yal Seo, Dong-Ha Lee, Kyung-Mee Lee, and Jeon-Young Lee. Discovery of schema information from a forest of selectively labeled ordered trees. In *Proc. of Workshop on Management of Semistructured Data (in Conjunction with PODS/SIGMOD)*, May 1997. http://www.research.att.com/~suciu/workshop-papers.html.

[14] Keishi Tajima, Yoshiaki Mizuuchi, Masatsugu Kitagawa, and Katsumi Tanaka. Cut as a querying unit for WWW, Netnews, and E-mail. In *Proc. of ACM Hypertext*, pages 235–244, Jun. 1998.