データベースシステム 119−39
(1 9 9 9. 7. 2 2)

あいまいな検索パターンの記述が容易な半構造データ検索言語

田 島 敬 史[†]

半構造データに対する検索を記述する場合，データの構造が不規則である点と，あらかじめ与えられたスキーマがない点が問題となる．これまでに提案されている半構造データのための検索言語では，これらの問題をワイルドカードを導入することで解決している．しかし，全体のデータ構造がわからない状態で検索を記述する場合，ワイルドカードが思わぬデータにマッチし，検索結果に不必要なデータまで含まれてしまうことがある．そこで本論文では，どのようなデータがより解として適当と思われるかの優先度を記述できる構文として、case 構文、smallest matching の構文、minimal matching の構文の三つを持つ検索言語を提案する．これらの構文を用いることにより，全体のデータ構造を知らない場合でも，必要なデータとだけマッチする検索文をより容易に記述できる．

Query Language Constructs for Uncertain Query Patterns in Semistructured Data

KEISHI TAJIMA[†]

Two main difficulties in specifying queries on semistructured data are structural heterogeneity of the data, and the lack of the schema in advance. In the languages proposed in past, wild cards are used to solve these problems. When we specify queries without the knowledge of the entire data structure, however, wild cards often match with unexpected data, and it causes noises in query answers. To solve this problem, we introduce constructs for specifying query patterns with order of likelihood to match with appropriate data. In this paper, we design three constructs, a case construct, a construct for smallest matching, and a construct for minimal matching. By using those constructs, we can specify queries returning only really expected answers more easily and with less knowledge on the entire data structure.

1. Introduction

Semistructured data are schema-less, self-describing data. Many researches have proposed slightly different data models for semistructured data, but all the recent proposals represent semistructured data by edge-labeled directed graphs[1)~3)].

For example, Figure 1 shows an example data described by the data model proposed in 1). This data represents a part of a movie database. The root node at the top of the figure is the entry point of the database, and it has references to all entries of movies and actors. Movie entries have two attributes title and cast, and actor entries have two attributes name and appear. Those attributes are represented by the edges outgoing from the root node of each entry. In this model, attribute values are also represented by labels of terminating edges.

Because of the lack of the rigid schema, subgraphs representing the same kind of data may have different structures. For example, the graphs beneath cast attributes of two movie entries have different structures. This structural heterogeneity makes set-oriented operations on those semantically homogeneous entities difficult. In addition, we may not know all those structures when writing queries because semistructured databases may come up with no predefined schema. This lack of schema also makes query writing difficult.

To solve these two problems, most query languages for semistructured data support path expressions including wild cards. For example, in UnQL[1)], a query "list the titles of all movies featuring an actor named Pitt" is described by using wild cards as follows[☆]:

† 神戸大学情報知能工学科 tajima@db.cs.kobe-u.ac.jp
Dept. of Comp. and Sys. Eng., Kobe University

☆ The syntax used here is slightly different form the original one.

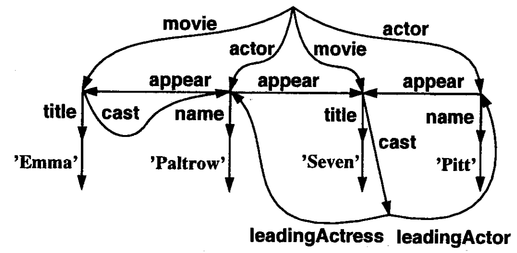

Fig. 1 An Example of Semistructured Data

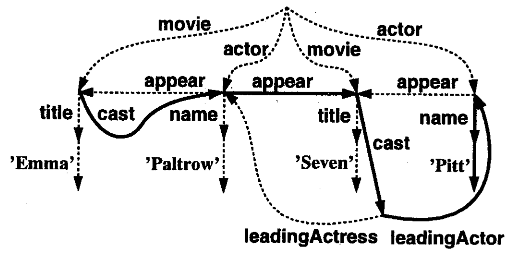

Fig. 2 An Unexpected Path

```
select l where
  movie⇒ {title⇒ l ⇒ {},
          cast⇒[_⇒]*name⇒'Pitt'⇒ {}}
```

where clause in the query describes a graph pattern to find. "_" 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 beneath cast attributes are heterogeneous, and the users are uncertain what structures may appear there. The expression in the where clause of this query matches with subtrees in the database that consist of an edge with a label movie from the root of the database, and two paths beneath it, one of which is starting with a title edge followed by some value *l*, and the other of which is a path of arbitrary length starting with a cast edge and ending with a name edge followed by a string value "Pitt." The query returns the set of the values *l* for all matching subtrees in the database.

In some cases, however, such wild cards that match with paths of arbitrary length happen to match with unexpected paths, and cause noises in query answers. For example, when we apply the query above to the database shown in Figure 1, the expression cast⇒[_⇒]*name⇒'Pitt'⇒ {} matches with an unexpected path shown in Figure 2, i.e. a path starting with cast edge from the entry of "Emma" to the entry of "Paltrow," going through appear edge to "Seven," going through cast and leadingActor edges to the actor "Pitt," and ending with the label "Pitt" in that entry. Therefore, the result of the query above includes the movie title "Emma" although Pitt does not appear in the movie "Emma."

One approach to avoid such unexpected matchings is to use more complicated regular expressions so that they eliminate such unexpected matchings. For example, the query below solves the problem:

```
select l where
  movie⇒ {title⇒ l ⇒ {},
          cast⇒[^appear⇒]*name⇒
              'Pitt'⇒ {}}
```

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

Specifying appropriate regular expressions that match only with really needed paths is, however, quite difficult task. Paths of arbitrary length may reach to everywhere in the database graph, and it is difficult to anticipate all "unexpected" cases. In order to anticipate all of them, we need to know the entire structure of the database, but it is not usually the case when we deal with semistructured data. In addition, even when we can anticipate all those cases, query specification excluding all those cases can be very complicated and not easy to read or write.

From the example above, we got two observations on query on semistructured data:

(1) In traditional databases, e.g. relational databases, when a schema and the description of needed information are given, a "correct" query is uniquely determined (except for its equivalences). On the other hand, when we query semistructured data associated with no rigid schema, there is no unique "correct" answer. We can only guess various queries that seem "appropriate," some of which return more correct answers and more noises, and other of which return less correct answers and less noises. This characteristic is similar to that of the information retrieval.

(2) Wild cards that match with arbitrary number of paths of arbitrary length are essentially dangerous, and not easy to use. We sometimes certainly need such wild cards, but such cases are rather rare, and more restricted usage of wild cards are more often needed. For example, the intention of the expres-

sion cast⇒[_⇒]*name⇒ in the first query is to skip some heterogeneous structure beneath the cast attributes, and is not to reach everywhere in the database graph through other actor entries and movie entries. This kind of usage, i.e. skipping some small heterogeneous structure, is indeed the most popular one.

Following these two observations, in this paper, we develop query language constructs in which various likely query patterns are specified with order of likelihood and wild cards match only with the best matching data. Currently we have designed three constructs: a case construct, a construct for smallest matching, and a construct for minimal matching. By using these constructs, we can specify queries returning only really expected answers more easily and with less detailed knowledge on the data structure.

Restricting wild cards to match only with the most likely data also has impact on the efficiency of query evaluation. The evaluation of queries including such wild cards may need more or less computation than the evaluation of queries with usual wild cards matching with all data satisfying the given condition. It depends on the indexing schemes and query evaluation techniques. In this paper, however, we do not discuss this issue any more. That is a future issue.

2. New Language Constructs

Now in this section, we informally explain the syntax and the semantics of our new constructs by using example queries. In this paper, we borrow UnQL as the base of our development. These constructs, however, can equally be developed on other languages for semistructured data. Here we omit the detailed explanation of UnQL for the brevity. Please refer to 1) for more detail.

2.1 A Construct for Specifying Likelihood Order

First, as the most basic construct to specify which patterns are most likely to be the data expected by the user, we introduce the following construct:

```
case condition, ..., condition
   | condition, ..., condition
       :
   | condition, ..., condition
end
```

which is used in where clause. This construct has a syntax similar to disjunctive conditions. Condition lists delimited by | specify alternative patterns. Each pattern, which in turn consists of multiple conditions, has its own scope of variables and can introduce new variables independently. However, outside this case construct, i.e. at other places in where clause or in select clause, only the variables introduced in all the alternative patters can be referred to. The difference from usual disjunctive conditions is that each alternative patterns are examined in left to right order, and only when there is no data instance that matches with a pattern, the next alternative is examined. By using this construct, we can specify the priority order of graph patterns that are likely to be the expected data. In this paper, we explain the semantics of this construct informally by using examples. For example, we can specify a query like below:

```
select $l_1$ where
  movie⇒ {title⇒ $l_1$ ⇒ {}, cast⇒ $t$} ←DB,
  case name⇒ $l_2$ ⇒ {} ← $t$
     | [_⇒]*name⇒ $l_2$ ⇒ {} ← $t$
  end,
  $l_2$='Pitt'
```

This example query is evaluated in the following way.

(1) Three conditions listed in where clause is evaluated in this order. The first condition, which is a pattern expression, is evaluated first. When a pattern expression is evaluated, variables introduced in that description, l_1 and t in this case, are bound to matching data instances. (In this case, edge labels for l_1 and subtrees for t.) For each pair of bindings of l_1 and t, an environment consisting of these two bindings is created.

(2) Under each environment created in the previous step, the second condition is evaluated. First, the pattern name⇒ l_2 ⇒ {} ← t is examined. For each data instance matching to it, an environment consisting of bindings for l_1, t, and l_2, which is only variable common to all the alternatives in the case construct, is created. If there is no data instance matching to the first pattern, the second alternative is examined in the same way.

(3) Under each environment created in the previous step, the third condition, which is a boolean formula, is evaluated. When a boolean formula is evaluated, only environments that let the formula be true is passed to the next step.

(4) Now all the conditions in the where clause have been evaluated. Then, under each environment passed from the previous step, the select clause is evaluated. The answer is the set

of the results of those evaluations.

When we issue this query on the example database shown before, it correctly answers {'Seven'} because when t is bound to the tree under cast attribute of "Emma," the pattern name⇒ l_2 ⇒ {} correctly matches with a path corresponding to the name of that cast, and the second pattern in the case construct is not used.

Note that the semantics of the query below is different from that of the query above:

select l_1 where
  movie⇒ {title⇒ l_1 ⇒ {}, cast⇒ t} ←DB,
  case name⇒'Pitt'⇒ {} ← t
    | [_⇒]*name⇒'Pitt'⇒ {} ← t
  end

This query returns {'Emma', 'Seven'} because the pattern name⇒'Pitt'⇒ {} matches with no path when t is bound to the tree under cast of "Emma." In the same way, the semantics of the query below is also different:

select l_1 where
  case movie⇒ {title⇒ l_1 ⇒ {},
                cast⇒name⇒ l_2 ⇒ {}}
         ←DB
     | movie⇒ {title⇒ l_1 ⇒ {},
                cast⇒[_⇒]*name⇒ l_2 ⇒ {}}
         ←DB
  end,
  l_2='Pitt'

This query returns {} because the first pattern matches with a path for the entry of "Emma" and its cast, and the second pattern is not used at all.

Writing queries like above using case construct requires the knowledge on what structures are most common in the data. On the other hand, the approach of writing conditions explicitly excluding the unneeded cases, such as [^appear]* explained before, requires the knowledge on what exceptional structures cause unwanted noises. In some cases, e.g. when there are many different kind of exceptional patterns, query specifications by the former approach may be more concise, and in some cases, those by the latter approach may be more concise. When we need to specify a query without any schema, or when we want to specify a query without reading the entire large schema, we can get the former kind of knowledge more easily. We can learn it by browsing relatively small portion of the data while in order to learn the latter kind of knowledge we need to browse larger portion of the data, or need "trial and error" with test queries. However, it is also difficult to write perfect query only with the former approach, and therefore, the combination of both approaches must be the best way.

2.2 A Construct for Shortest Matching

By using case construct, we can specify a variety of order of likelihood. Some kind of order, however, cannot be specified by case construct with finite number of alternative patterns. One typical example is a case where we give priority to shorter paths. In such a case, we need an expression of infinite length like case _⇒a⇒ t | _⇒_⇒a⇒ t | _⇒_⇒_⇒a⇒ t | ...end. Because situations where we want to give priority to shorter paths is very popular, we introduce a construct below for those cases:

  (<) *condition*, ...,*condition* end

This expression selects only the shortest graph pattern(s) among those that satisfy the listed conditions. The part that should be shortest is designated by ⌊⌋ construct in condition specifications. We explain the semantics of this construct only informally by using an example. For example, by using this construct, the first query in Introduction can be rewritten as below:

select l_1 where
  movie⇒ {title⇒ l_1 ⇒ {}, cast⇒ t} ←DB,
  (<) ⌊[_⇒]*⌋name⇒ l_2 ⇒ {} ← t end,
  l_2='Pitt'

This query is evaluated in the following way:

(1) The first condition is evaluated, and environments are created in the same way as we did for the query in the previous subsection.

(2) For each environment created in the previous step, the second condition is evaluated. It collects all graphs matching the pattern, and select the one(s) in which the path matching to the part enclosed in ⌊⌋ is the shortest. Then environments including bindings of newly introduced variables are created. If there are more than one ⌊⌋ constructs, the sum of the length of those parts is compared.

(3) For each environment created in the previous step, the third condition is examined in the same way as we did for the previous query.

This query is better than the previous one using case construct. The previous query returns an answer including noises if there is a movie entry that has cast attribute with a structure similar to the one in the entry of "Seven," and that movie is not featuring "Pitt" but some actor appearing in another movie featuring "Pitt." The query above, however, does not include those noises in the answer. In this way, although the case construct can specify wide range of likelihood order, we can often write a better query by the construct for shortest paths.

Note again that the following two queries have different semantics from the query above:

select l_1 where
  (<) movie⇒
    {title⇒ l_1 ⇒ {},
     cast⇒ ⌊[_⇒]*⌋name⇒ l_2 ⇒ {}} ←DB
  end,
  l_2='Pitt'

which returns {}, and

select l_1 where
  movie⇒ {title⇒ l_1 ⇒ {}, cast⇒ t} ←DB,
  (<) ⌊[_⇒]*⌋name⇒'Pitt'⇒ {} ← t end

which returns {'Emma', 'Seven'}.

The semantics of the construct for shortest paths can naturally be generalized for smallest graphs. When ⌊⌋ construct is used for a graph expression, it select only the smallest graph(s) in size, i.e. in the number of edges in it. For example consider the query below:

select {name⇒ l_1, title⇒ l_2} where
  (<) [_⇒]*⌊{[_⇒]*name⇒ l_1 ⇒ {}
      [_⇒]*title⇒ l_2 ⇒ {}}⌋ ← DB
  end

This query finds at arbitrary depth the smallest trees that include labels name and title. When the user has almost no knowledge on the structure of the database, one may try this query to retrieve pairs of an actor name and a movie title featuring that actor. This query does not always succeed, but when this query is issued on the example database shown before, it fortunately returns the correct answer.

2.3 A Construct for Minimal Matching

In just the same way we have introduced a construct for smallest graphs, we also introduce a construct for minimal graphs shown below:

(⊂) *condition*, ..., *condition* end

While (<) construct selects the smallest one(s) in the sense of the number of edges, (⊂) construct selects the minimal one(s) in the sense of supergraph-subgraph relationship. By using this construct, now we can write the query "list the titles of all movies featuring Pitt" as follows:

select l_1 where
  movie⇒ {title⇒ l_1 ⇒ {}, cast⇒ t_1} ←DB,
  (⊂) actor⇒ t_2 ← DB,
    ⌊[_⇒]*⌋t_2 ← t_1 end,
  name⇒'Pitt'⇒ {} ← t_2

This query is evaluated in the same way as queries with (<) construct except that not the smallest but the minimal graphs are selected.

This query is better than the previous ones. In this query, the pattern ⌊[_⇒]*⌋t_2 matches only with the first actor entry on each path beneath cast edge. Therefore, as long as there is not a path from cast edge to some actor entry who does not appear in that movie without going through other actor entries, this query returns the correct answer.

Unlike the previous examples, the semantics of the query below is identical to that of the query above.

select l_1 where
  (⊂) actor⇒ t_2 ← DB,
    movie⇒ {title⇒ l_1 ⇒ {},
      cast⇒⌊[_⇒]*⌋t_2} ←DB end,
  name⇒'Pitt'⇒ {} ← t_2

On the other hand, the query below is not identical to those above:

select l_1 where
  actor⇒ t_2 ← DB,
  (⊂) movie⇒ {title⇒ l_1 ⇒ {},
      cast⇒⌊[_⇒]*⌋t_2} ←DB end,
  name⇒'Pitt'⇒ {} ← t_2

This query evaluate the second condition for each t_2, i.e. for each actor entry. Therefore, for each movie entry, if there is at least one path from that movie entry to the entry of "Pitt," the second condition extract the minimal paths among them. As a result of it, this query returns {'Emma', 'Seven'}.

3. Related Work

The case construct is expressible by Datalog with stratified negation, GraphLog[4)], or by the traverse construct of UnQL, although not in concise forms. A graph query language proposed in 5) supports rewrite construct, which is similar to traverse construct. It can also express the case construct. 5) also introduces a construct to find shortest-paths. Although 5) does not define the semantics of their construct in detail, it seems that their construct can express only special cases of those expressible by our construct. A language proposed in 6) support node and edge deletion operations. However, constructs for smallest or minimal matching may or may not be expressible by using these deletion operations. The comparison of our language with those languages or some other languages, such as 7), is not fully examined yet. The comparison of the expressive power of our language with that of linear logic, or the class of computable functions must also be interesting. All these are future issues.

In the previous research, as another solution for the problem discussed in Introduction, we have proposed constructs to distinguish edges within entities and those across entities in semistructured data[8]. We consider entities in semistructured data are represented by the rooted subtrees consisting of only exclusive references, i.e. edges that are only edges referring to their destination nodes. By this criterion, the example data in Figure 1 is correctly divided into two actor entities and two movie entities. By using the constructs proposed in 8), the query "list the titles of all movies featuring "Pitt" is described as follows:

select l_1 where
  movie⇒{title⇒ l_1 ⇒ {},
            [cast⇻ | cast⇛[_⇛]*_⇻]
                name⇒'Pitt'⇒ {}}←DB

[... | ...] is a disjunction expression. *label*⇛ is an expression that matches only with edges within an entity, in other words, only with exclusive references. On the other hand, *label*⇻ is an expression that matches only with edges across entities, in other words, only with non-exclusive references. This query try to find movie entry that are directly connected to the entry of "Pitt" through a path including only one edge acrossing the boarder of entities.

In that research, we use exclusiveness of references to distinguish edges within one entity and edges across entities. This method is, however, not perfect, of course. Therefore, the following query that uses (<) construct together with those edge expressions is better than the query above.

select l_1 where
  (<) movie⇒{title⇒ l_1 ⇒ {},
            ⌊[cast⇻ | cast⇛[_⇛]*_⇻]⌋
                name⇒'Pitt'⇒ {}}←DB

Note that ⌊⌋ construct must enclose the entire disjunction expression. If ⌊⌋ construct appeared only either of these two alternative patterns in the disjunction, the semantics of this (<) construct would be ambiguous.

4. Conclusion and Future Issues

In this paper, we propose three constructs for query language for semistructured data: a case construct, a construct for smallest matching, and a construct for minimal matching. By using these constructs, we can specify queries returning only really needed answers with less knowledge on the entire data structure.

This paper shows only preliminary ideas, and there are many future issues, such as:

- the decision problem of equivalency or containment between queries,
- efficient indexing and evaluation schemes for those constructs, and
- the comparison of the expressive power with other proposed languages.

Acknowledgments The author 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) Buneman, P., Davidson, S., Hillebrand, G. and Suciu, D.: A Query Language and Optimization Techniques for Unstructured Data, *Proc. of ACM SIGMOD*, pp. 505–516 (1996).

2) Abiteboul, S., Quass, D., McHugh, J., Widom, J. and Wiener, J.L.: The Lorel Query Language for Semistructured Data, *International Journal of Digital Libraries*, Vol. 1, No. 1, pp. 68–88 (1997).

3) Fernández, M.F., Florescu, D., Kang, J., Levy, A. and Suciu, D.: Catching the Boat with Strudel: Experiences with a Web-Site Management System, *Proc. of ACM SIGMOD*, pp. 414–425 (1998).

4) Consens, M. P. and Mendelzon, A. O.: GraphLog: A Visual Formalism for Real Life Recursion, *Proc. of ACM PODS*, pp. 404–416 (1990).

5) Güting, R. H.: GraphDB: Modeling and Querying Graphs in Databases, *Proc. of VLDB*, pp. 297–308 (1994).

6) Gyssens, M., Paradaens, J. and Van Gucht, D.: A Graph-Oriented Object Database Model, *Proc. of ACM PODS*, pp. 417–424 (1990).

7) Neven, F. and Van den Bussche, J.: Expressiveness of Structured Document Query Languages based on Attribute Grammars (exteded abstract), *Proc. of ACM PODS*, pp. 11–17 (1998).

8) Tajima, K.: Querying Composite Objects in Semistructured Data, *Proc. of FODO* (1998).