ELSEVIER

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Data & Knowledge Engineering 46 (2003) 225–246

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Techniques for the evaluation of XML queries: a survey ☆

Tae-Sun Chung *, Hyoung-Joo Kim

School of Computer Science and Engineering, Seoul National University, San 56-1, Shillim-dong, Gwanak-gu, Seoul 151-742, South Korea

Received 31 October 2001; accepted 13 March 2002

Abstract

As XML has become an emerging standard for information exchange on the World Wide Web it has gained great attention among database communities with respect to extraction of information from XML, which is considered as a database model. XML queries enable users to issue many kinds of complex queries using regular path expressions. However, they usually require large search space during query processing. So, the problem of XML query processing has received significant attention. This paper surveys the state of the art on the problem of XML query evaluation. We consider the problem in three dimensions: XML instance storage, XML query languages and XML views, and XML query language processing. We describe the problem definition, algorithms proposed to solve it and the relevant research issues.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Semistructured data; XML; Query processing; Database

1. Introduction

Recently, XML [4] has become an emerging standard for information exchange on the World Wide Web. It has gained great attention among database communities with respect to extraction of information from XML, which is considered as a database model. That is, as XML is self-describing, we can issue many kinds of queries against XML documents in heterogeneous sources and get the necessary information.

☆ This work was supported by the Brain Korea 21 Project.
* Corresponding author.
E-mail addresses: tschung@papaya.snu.ac.kr (T.-S. Chung), hjk@papaya.snu.ac.kr (H.-J. Kim).

0169-023X/03/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0169-023X(02)00211-2

In this paper, we survey the state of the art in XML query processing and indexing/storage techniques. We consider the problem in three dimensions: XML instance storage, XML query languages and XML views, and XML query language processing.

First, there are two kinds of approaches to storing XML documents. One is using special purpose query engines for semistructured data since an XML document can be regarded as an instance of a semistructured data set. That is, data in XML is mapped to a semistructured data model and the query processor based on graph traversal processes XML queries.

The other is using traditional databases such as relational databases or object-oriented databases for storing and querying XML documents. This has the advantage of using the traditional database engines. However, as there exist differences between XML data model and traditional database ones, there are problems of impedance mismatch.

We classify the techniques of using commercial database systems to store and query XML documents into two types. One is to store the graph to which XML data is mapped. It is simple and has an advantage that it works in the absence of DTDs. However, complex queries are constructed to query XML documents and many join operations are required to process the queries.

The other is inferring schemata from the DTDs of the XML documents. In this case, issues on how to store XML data that has variability in structural characteristic should be addressed.

In the context of XML query languages and XML views, we classify the works based on whether the queries and views are composed of single or multiple regular path expressions. We present the problem definition and review the query rewriting problem.

Finally, in the context of XML query processing, we classify the techniques into two types. The first category is the schema extraction technique. That is, given a particular data instance of large size, the technique finds the schema for it and traverses the schema graph of small size instead of the data graph. Graph schemas [16,35], DataGuides [19,29], T-index [27] and so on belong to this group.

Second, the graph pruning technique is proposed. It restricts search to a fragment of the graph by adding information to each object. NodeInfo [11], MergeNodeInfo [11], SigDAQ [32] and so on are classified to be in this group.

In addition, stream-based query answering methods such as X-scan [22] are proposed. For example, X-scan matches regular path expressions and returns results in pipelined fashion. The stream-based query answering methods are beyond the scope of this paper.

2. Basic concepts

2.1. Semistructured data model

In this paper, we use the object exchange model (OEM) [30] which is the most representative among semistructured data models.

Data in the OEM model is represented as an edge-labeled graph. Every object in OEM consists of an identifier and a value, and the nodes in the graph are objects and edges are labeled with attribute names. The OEM objects are classified as the following two kinds of objects, depending on their values.

- Atomic objects: The value of the atomic objects is an atomic quantity, such as an integer, string, image, sound and so on.
- Complex objects: The value of the complex objects is a set of <label, id> pairs.

The right side of Fig. 1 shows an OEM graph. Here, &0, &1, etc. are object identifiers. Objects such as &12 and &13 are atomic objects and those such as &1 and &2 are complex objects.

2.2. XML

XML (eXtensible Mark-up Language) is an emerging standard for information exchange on the World Wide Web. An XML document is composed of the optional DTD which describes the document's structures and data encoded in XML. Data in XML is composed of sequences of elements which can be nested and may have attributes. For example, Fig. 2 shows a DTD specification.

Here, the second line states that an element *person* has the *name* and *address* subelements, and he or she has zero or more vehicles, and finally, is a student or a company employee. The left side of Fig. 1 shows an XML document that conforms to the DTD in Fig 2.

XML corresponds closely to semistructured data based on edge-labeled graph, so it is possible to map data in XML to a semistructured data model such as OEM. Before mapping data in XML to an OEM graph, we assume that no element has attributes other than the attribute `ID` and the attribute `IDREF` [4]. The XML elements which have attributes other than those mentioned above can be redefined as ones that do not have them in the following manner [34].

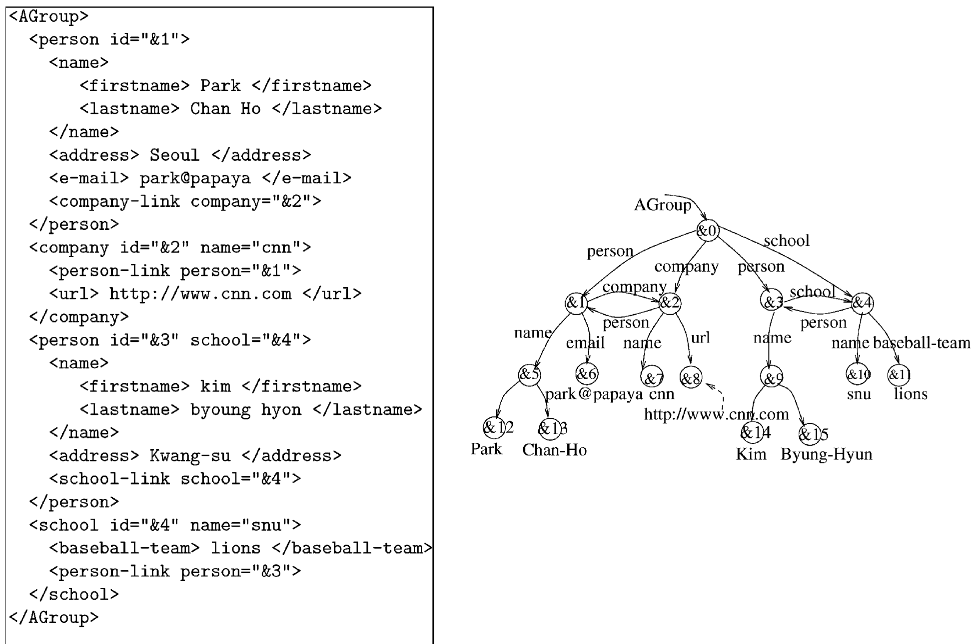
```
<Paper format="ps">
  <author> Serge Abiteboul </author>
</Paper>
```

This is converted to the following XML data.

```
<Paper>
  <format> ps </format>
  <value>
    <author> Serge Abiteboul </author>
  </value>
</Paper>
```

Data in XML can be represented by the OEM model. [1] That is, XML elements are represented by nodes of an OEM graph and element–subelement, element–attribute and reference relationships are represented by edges labeled by the corresponding names. Values of XML data are represented by leaves in the OEM graph. Fig. 1 shows an XML data instance and a corresponding OEM graph.

[1] In this case some information can be lost.



```
<AGroup>
  <person id="&1">
    <name>
       <firstname> Park </firstname>
       <lastname> Chan Ho </lastname>
    </name>
    <address> Seoul </address>
    <e-mail> park@papaya </e-mail>
    <company-link company="&2">
  </person>
  <company id="&2" name="cnn">
    <person-link person="&1">
    <url> http://www.cnn.com </url>
  </company>
  <person id="&3" school="&4">
    <name>
       <firstname> kim </firstname>
       <lastname> byoung hyon </lastname>
    </name>
    <address> Kwang-su </address>
    <school-link school="&4">
  </person>
  <school id="&4" name="snu">
    <baseball-team> lions </baseball-team>
    <person-link person="&3">
  </school>
</AGroup>
```


Fig. 1. An example of XML data and OEM graph.

```
<!ELEMENT AGroup (person|school|company|vehicle)+>
<!ELEMENT person (name, address, vehicle*,
       (school-link|company-link))>
<!ELEMENT name (firstname?, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT vehicle (model, company-link, gear?)>
<!ELEMENT school-link EMPTY>
<!ATTLIST school-link school IDREF #IMPLIED>
<!ELEMENT company-link EMPTY>
<!ATTLIST company-link company IDREF #IMPLIED>
<!ELEMENT model (#PCDATA)>
<!ELEMENT gear (#PCDATA)>
<!ELEMENT school (name, baseball-team?, person-link+,url?)>
<!ATTLIST school name CDATA #CDATA REQUIRED>
<!ELEMENT baseball-team (#PCDATA)>
<!ELEMENT person-link EMPTY>
<!ATTLIST person-link person IDREF #IMPLIED>
<!ELEMENT url (#PCDATA)>
<!ELEMENT company (name, person-link+, url?)>
<!ATTLIST company name CDATA #CDATA REQUIRED>
<!ELEMENT alumni (name, year, school-link)>
<!ATTLIST alumni name CDATA #CDATA REQUIRED>
<!ELEMENT year (#PCDATA)>
```

Fig. 2. An example DTD.

2.3. Query language

XML query languages are derived from those of object-oriented DBMSs such as OQL [7] and XSQL [23]. In these object-oriented database query languages, the presence of path expressions provides syntactic sugar for long join sequences. Path expressions describe paths along the class composition hierarchy and are defined as follows.

Definition 1 (*Path expression*). A path expression is of the form

$$\text{selector}_0.\text{Att}_1\{[\text{selector}_1]\}.\ldots.\text{Att}_m\{[\text{selector}_m]\}$$

where $m \geqslant 0$, and $\text{selector}_1, \ldots, \text{selector}_m$ are optional terms. The selector_k $(k = 0, 1, \ldots, m)$ is an object id, a variable that ranges over object ids, or object values. The Att_k $(k = 1, 2, \ldots, m)$ is either an attribute name or an attribute variable that ranges over attribute names.

Example 1. The following query retrieves all persons' schools that have the url 'http://www.snu.ac.kr'.

```
select Y
from Person X
where X.School[Y].url["http://www.snu.ac.kr"]
```

XML query languages are based on the following regular path expressions that expand the path expressions. For instance, XML-QL [13], UnQL [5], Lorel [1] and so on are query languages based on the regular path expressions.

Definition 2 (*Regular path expression*). A regular path expression is a form of H.P where

1. H is an object name or a variable denoting an object,
2. P is a regular expression over labels in an OEM graph. Namely, P = label|(P|P)|(P.P)|P∗.

Example 2. By using regular path expressions, the following query can be issued even though one does not know the database schema.

```
select P.name
from person P, P._∗.url U
where U = "http://www.snu.ac.kr"
```

The query retrieves all names of persons when there is a path that has the edge *person* followed by any sequence of arbitrary edges, and next, have the edge url with the value of "http://www.snu.ac.kr".

For simplicity, we use the regular path expression throughout the paper and it can cover most of techniques for evaluation of XML queries. However, the regular path expression does not

represent conditions on siblings. On the other hand, XQuery [8] which embeds XPath [3] can represent conditions on siblings and an index structure for XPath queries in proposed in [21]. In addition, a formal framework for addressing orderedness in XML documents is studied in [28].

3. XML instance storage

In this section we address XML instance storage. We classify the techniques of XML instance storage into two categories. The one is using native XML databases and the other is using commercial database systems.

Lore [25] can be considered as a native XML database management system for XML. The data model used in Lore is OEM which is presented in Section 2.1. XML instance is stored as graph form in Lore.

As a commercial product, native XML instance includes exelon [15] and Tamino [2]. They provide persistence for XML data and query languages. In Section 3.1 we discuss XML instance storage using commercial database management systems.

3.1. Relational and object-oriented databases

There have been many activities dealing with storing and querying XML documents using relational or object-oriented database systems. In this case, since we can use query engines of traditional database systems, the key point is how to create database schemas to store XML documents. We classify the techniques into two groups: One is using relational database systems and the other is using object-oriented database systems.

3.1.1. Storing XML documents using a relational database system

Since relational database systems are widely used database systems, many researchers have discussed the problem of storing and querying XML data using a relational database system. We classify the techniques into two categories. One strategy [17] is to store the graph to which XML data is mapped. Another option [14,33] is to infer relational database schemas to store the XML documents.

3.1.1.1. Graph based approach. In the graph based approach, XML data is represented as a graph and the graph itself is mapped into relational tuples.

The simplest scheme is to use edge tables and value tables. In this section we will consider the basic scheme that uses edge tables and value tables for storing XML documents. The edge table stores the oids of the source and target objects of each edge of the graph, the label of the edge, an ordinal number representing the order of edges, and a flag that shows whether the edge is an internal node or a leaf node. The value table stores values (i.e. strings) of XML documents. So, it has the field vids storing oids of values and values storing all strings. [2]

[2] We assume that all XML data types are strings.

For example, if we store the MLB database in Fig. 8 in a relational database system, the relational table is constructed as follows. Here, assume that the vids of the leaf nodes 3, 4, and 5 are v3, v4, and v5 respectively.

Edge table

Source	Target	Label	Ordinal	Flag
root	1	MLB	1	Ref
1	2	National	1	Ref
1	14	National	2	Ref
1	24	American	3	Ref
2	v3	Division	1	String
2	v4	Stadium	2	String
2	v5	Name	3	String

Value table

vid	Value
v3	West
v4	Dodgers stadium
v5	LA Dodgers

The queries over semistructured data are converted to SQL queries. For example, consider the following Lorel-like query over semistructured data that asks for the stadiums of national league teams in the MLB.

```
select X
from MLB.National.stadium X
```

The query is converted to the following SQL query.

```
select v.value
from Edge e1, Edge e2, Edge e3, Val v
where e1.source = root and e1.label = MLB
  and e1.target = e2.source and e2.label = National
  and e2.target = e3.source and e3.label = stadium
  and e3.target = v.vid
```

3.1.1.2. Schema extraction. Another approach is to infer relational database schemas from the DTDs of XML documents. Although the DTD is an optional feature of XML, the DTD can be inferred from XML data by the technique proposed in [18]. Similarly, the authors in [14] provided a technique of storing semistructured data in the absence of DTDs. They provided a query

language named STORED that specifies the mapping between the semistructured data model and the relational model.

Here, we will review the key idea underlying the algorithm in [33]. First, a DTD graph is introduced. The DTD graph represents the structure of a DTD. Formally, a DTD graph $G_D = (V, E)$ is a graph where V is a set of nodes, i.e. elements, attributes and operators. $E \subseteq V \times V$ is a set of edges representing relationships between nodes. For the set of nodes V, each element appears exactly once, while attributes and operators appear as many times as they appear in the DTD. Here, the DTD is a simplified one [33]. That is, the binary operators (',' or '|') do not appear inside any of the operators. Fig. 3 shows an example DTD graph.

Next, we decide what relations to create from the DTD graph, based on the following rules.

1. Relations are created for element nodes that have an in-degree of zero. Otherwise, the element cannot be reached. For example, the relation for "alumni" is created, because the element node *alumni* has an in-degree of zero.
2. Elements below a '$*$' or a '$+$' node are made into separate relations. This is necessary for creating a new relation for a set-valued child. For instance, the element node *vehicle* that is below a '$*$' node is made into a separate relation.
3. Nodes with an in-degree of one are inlined. For example, in Fig. 3, nodes *gear* or *model* are inlined as they have in-degree of one.
4. Among mutually recursive elements all having in-degree one, one of them is made into a separate relation.
5. Element nodes having an in-degree greater than one are made into separate relations.

If we apply these five rules to the DTD graph in Fig. 3, the relations *Person*, *Vehicle*, *Company*, *School*, *Alumni*, and *Url* will be created. Once we decide which relations are created, we construct a relational schema. In the DTD graph, if X is an element node that is made into a separate relation, it inlines all the nodes Y that are reachable from it such that there is no node that is made into a separate relation in the path from X to Y. A relational schema is created for the DTD graph in Fig. 3 as follows.

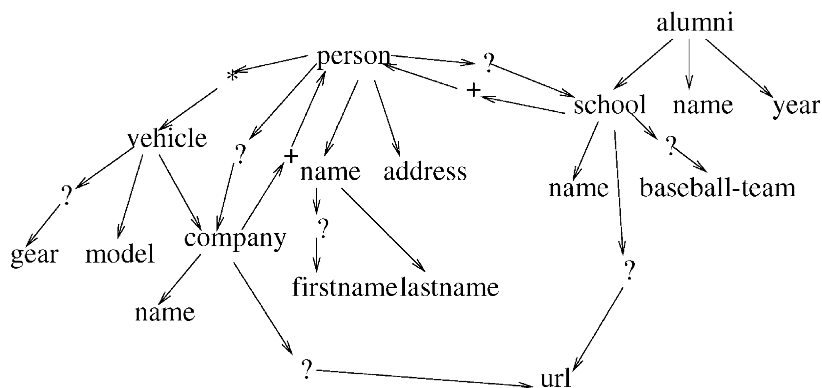


Fig. 3. A DTD graph.

```
person(personID: integer, person.ParentID: integer, person.parent-
CODE: integer,
  person.name.firstname: string, person.name.lastname: string,
  person.address: string)
vehicle(vehicleID: integer, vehicle.parentID: integer,
  vehicle.parentCODE: integer, vehicle.gear: string, vehicle.model:
string)
company(companyID: integer, companyparentID: integer,
  company.parentCODE: integer, company.name: string)
school(schoolID: integer, school.parentID: integer, school.parent-
CODE: integer,
  school.name: string, school.baseball-team: string)
alumni(alumniID: integer, alumni.name: string, alumni.year: string)
url(urlID: integer, url.parentID: integer, url.parentCODE: integer)
```

The ID field that each relation has serves as the key of that relation and relations corresponding to element nodes that have a parent also have a parentID field that serves as a foreign key. For example, the relation vehicle has a foreign key vehicle.parentID that joins vehicles with persons.

Semistructured queries are processed as follows. For instance, consider the following Lorel-like query over semistructured data.

```
select X.name.firstname, X.name.lastname
from Person X, X.vehicle Y
where Y.model = "SM 5"
```

The query is converted to the following SQL query.

```
select P."person.name.firstname", P."person.name.lastname"
from person P, Vehicle V
where P.personID = V."vehicle.parentID" and V."vehicle.model" = "SM 5"
  and V."vehicle.parentCODE" = 0
```

Here, the expression V."vehicle.parentCODE" = 0 indicates that the parent of the vehicle is a person.

3.1.2. Storing XML Documents in an Object-oriented database systems

Like in the case of relational database systems, the techniques of storing XML documents using an object-oriented database system are classified in two groups. One [15] is to store the graph to which XML data is mapped. The other [9,12] is to infer object-oriented database schemas from the DTDs of the XML documents. In the graph based approach, there is no special difference between relational and object-oriented approaches. So, we will consider the technique of extracting object-oriented schemas from XML DTDs.

Authors in [9] suggest a technique of processing SGML data using an OODBMS. They derive an object-oriented schema by using the DTD. That is, each class is derived from each element declaration. The choice operator ('|') is modeled by an union type and the occurrence indicators ('+' or '*') are represented by lists. Values (e.g. strings) of XML data are represented by O_2 classes of appropriate content types (e.g. text) using inheritance. Fig. 4 shows an object-oriented schema for the DTD in Fig. 2. As an extension to the work in [9], the technique of extracting object-oriented schemas using inheritance has been proposed in [12]. As we considered in Section 5, we can classify the person element into four categories: (1) ones who have one or more vehicles and work for companies, (2) ones who have no vehicle and work for companies, (3) ones who have one or more vehicles and are students and (4) ones who have no vehicle address and are students. The classification information can be used for designing object-oriented schemas by means of inheritance semantics. In the above example, each group is defined as *Person1*, *Person2*, *Person3*, and *Person4* type classes that inherit the general class *Person*. Here, for example, as *Person1* is a specialization of *Person*, the inheritance semantics is satisfied.

If we design object-oriented schemas in this way, it can be used for enhancing query evaluation. For example, if a query is related to students having vehicles, a query processor can only traverse extents of *Person3*.

Like in the relational database case, queries over semistructured data can be converted to object-oriented database query. For example, consider the following Lorel-like query over semistructured data.

```
select X.name.firstname, X.name.lastname
from person X, X.vehicle Y
where X.address = "Seoul", Y.model = "EF-Sonata", Y.gear = "auto"
```

The query asks for the first and last name of the person who has a vehicle "EF-Sonata" with an automatic transmission. The query is converted to the following object-oriented database query

```
class Person public type tuple(name:Name, address:Address,
       vehicle:list(Vehicle),union(school:School,company:Company))
class School public type tuple(name:string,
       baseball-team:Baseball-team,person:list(Person),url:Url)
class Company public type tuple(name:string,person:list(Person),
       url:Url)
class Vehicle public type tuple(model:Model,company:Company,
       gear:Gear)
class Alumni public type tuple(name:string,year:Year,
       school:School)
class Name public type tuple(firstname:Firstname,lastname:Lastname)
class Firstname inherit Text
class Lastname inherit Text
class Address inherit Text
class Baseball-team inherit Text
class Url inherit Text
class Model inherit Text
class Gear inherit Text
class Year inherit Text
```

Fig. 4. An OODB schema.

language. Here, as the variable p bound to the class *Person* has an attribute *vehicle*, only the instances of the class *Person1*, *Person2* are traversed.

```
select tuple(f: p."name.firstname", l: p."name.lastname")
from p in (Person1 or Person2), y in (p.Vehicle and Vehicle1)
where p.address = "Seoul", y.model = "EF-Sonata", y.gear = "auto"
```

4. XML query languages and XML views

We classify the problem of answering XML queries into two categories according to whether the queries and views are composed of single or multiple regular path expression. In this section, we address the problem definition and present query rewriting using views.

4.1. Single regular path expression

4.1.1. Problem definition

We define single regular path queries as follows.

Definition 3 (*Single regular path query*). Given a regular path expression r and a data graph D which is an OEM edge-labeled graph, the result of r on D is the set of objects on D that are reachable by the regular path expression r.

The result of single regular path query is computed as follows. First, construct some automaton A that is equivalent to the regular path expression r. And then, traverse the automaton A from the initial state to the final state while keeping the corresponding nodes in the data graph. And finally, retrieve the set of nodes corresponding to the final state of the automaton A. Algorithm 1 shows this process.

Algorithm 1. Computing simple regular path query

1: **Input:** A regular path expression r and a data graph D
2: **Output:** the set of object ids on D that are reachable by the regular path expression r
3: **Procedure** Query-Evaluation(r, D)
4: Construct an automaton A corresponding to r;
5: Closure $= \{(x_1, s_1)\}$ where $\{x_1, x_2, \ldots,\}$ is the set of nodes in D, with x_1 being the root and $\{s_1, s_2, \ldots,\}$ is the set of states in A with s_1 being the start state;
6: **while** Closure has not reached a fixpoint **do**
7:    **if** for some $(x, s) \in$ Closure there exist $(x \xrightarrow{a} x')$ in D and $(s \xrightarrow{a} s')$ in A **then**
8:       Closure $+= (x', s')$;
9:    **end if**
10: **end while**
11: $p \leftarrow$ the set of nodes x such that Closure contains some pair (x, s), with s a terminal state in A;
12: return p;

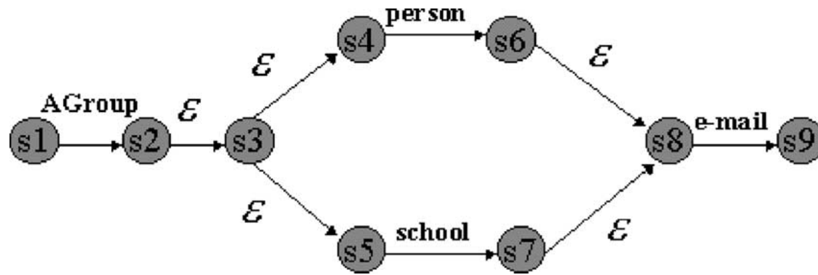


Fig. 5. The NFA for the regular path expression.

Table 1
The change of the set *Closure*

Iteration	Closure
0	$\{(\text{root}, s1)\}$
1	$\{(\text{root}, s1), (\&0, s2), (\&0, s3), (\&0, s4), (\&0, s5)\}$
2	$\{(\text{root}, s1), (\&0, s2), (\&0, s3), (\&0, s4), (\&0, s5), (\&1, s6), (\&1, s8), (\&3, s6), (\&3, s8), (\&4, s7), (\&4, s8)\}$
3	$\{(\text{root}, s1), (\&0, s2), (\&0, s3), (\&0, s4), (\&0, s5), (\&1, s6), (\&1, s8), (\&3, s6), (\&3, s8), (\&4, s7), (\&4, s8), (\&6, s9)\}$

Example 3. Suppose the regular path expression 'AGroup.(person|school).email' and the data graph in Fig. 1 are given. The non-deterministic automaton corresponding to the regular path expression is shown in Fig. 5. Table 1 shows the change of the set *Closure* in the while loop in Algorithm 1. After three iterations, the set *Closure* reaches a fixpoint. As the state $s9$ is a final state, the object id &6 is returned.

In addition, authors in [20] provide a technique that uses a lazy DFA for XML stream processing. The lazy DFA is constructed at run time, on demand. Authors show that only a small set of the DFA states need to be computed.

4.1.2. Rewriting of single regular path queries

The problem is computing the rewriting of a regular expression E_0 in terms of other regular expressions $\varepsilon = \{E_1, \ldots, E_n\}$. The authors in [6] showed that the problem of checking whether there is a non-empty rewriting is EXPSPACE-complete and provided an optimal 2-EXPSPACE algorithm for computing the rewriting.

Algorithm 2 shows the proposed rewriting algorithm. Let $re(e)$ denote the regular expression associated to the symbol e. First, the algorithm constructs a deterministic automaton A_d such that $L(A_d) = L(E_0)$. Next, it constructs the automaton $A' = (\sum_{\varepsilon}, S, s_0, \delta', S - F)$ where $s_j \in \delta'(s_i, e)$ if and only if $\exists_w \in L(re(e))$ such that $s_j \in \delta^*(s_i, w)$.

Algorithm 2. Rewriting of single regular path queries

1: **Input:** A regular path query E_0, a set of views $\varepsilon = \{E_1, \dots, E_n\}$
2: **Output:** An automaton R_{ε,E_0} which represents the rewriting
3: **Procedure** Query-Rewriting(E_0, ε)
4: Construct an automaton $A_d = (\sum, S, s_0, \delta, F)$ such that $L(A_d) = L(E_0)$;
5: Construct an automaton $A' = (\sum_\varepsilon, S, s_0, \delta', S - F)$ where $s_j \in \delta'(s_i, e)$ if and only if $\exists_w \in L(re(e))$ such that $s_j \in \delta^*(s_i, w)$;
6: Return $R_{\varepsilon,E_0} = \overline{A}'$;

Here, if A' accepts a $\sum_\varepsilon$-word $e_1, \dots, e_n$, then there exist n $\sum$-words $w_1, \dots, w_n$ such that $w_i \in L(re(e_i))$ $(i = 1, \dots, n)$ and such that the $\sum$-word $w_1, \dots, w_n$ is rejected by A_d. On the other hand, if there exists n $\sum$-words $w_1, \dots, w_n$ such that $w_i \in L(re(e_i))$ $(i = 1, \dots, n)$ and such that the $\sum$-word $w_1, \dots, w_n$ is rejected by A_d, then the $\sum_\varepsilon$-word $e_1, \dots, e_n$ is accepted by A'. Hence, R_{ε,E_0} which is the complement of A' accepts a $\sum_\varepsilon$-word $e_1, \dots, e_n$ if and only if all $\sum$-words $w_1, \dots, w_n$ such that $w_i \in L(re(e_i))$ $(i = 1, \dots, n)$ are accepted by A_d.

Example 4. Consider the query E_0 = AGroup.(person.company) + name.firstname and the set of views ε = {AGroup, person.company, name.firstname}. Here, $re(e_1)$ = AGroup, $re(e_2)$ = person. company, and $re(e_3)$ = name.firstname. Fig. 6(a)–(c) show A_d, A', and $\overline{A}'$ which are constructed in each step of Algorithm 2.

4.2. Multiple regular path expression

4.2.1. Problem definition

User's queries are usually composed of several regular path expressions. For example, consider the following query.

```
select p
from DB.movie p, p.actor.(Tom Cruise|Brad Pitt)q
where p.year = 2000
```

The query asks for movies which have actors 'Tom Cruise' or 'Brad Pitt' and were produced in the year 2000. The query consists of three regular path expressions, namely, DB.movie p,

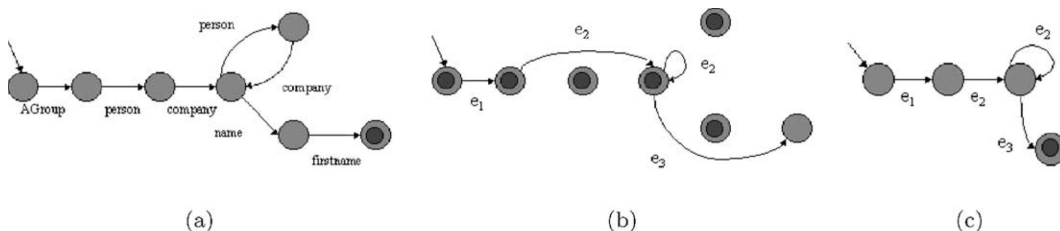


Fig. 6. Rewriting of a single regular path query: (a) An automaton A_d; (b) an automaton A'; (c) an automaton $\overline{A}'$.

p.actor.(Tom Cruise|Brad Pitt)q, and p.year = 2000. In processing XML queries, it is an important issue whether query optimization techniques can process queries that have more than one regular path expression. So, we define multiple regular path queries as follows.

Definition 4 (*Multiple regular path query*). A multiple regular path query is an expression of the form $q(\mathbf{x}) : -y_0 r_0 z_0, \ldots, y_{n-1} r_{n-1} z_{n-1}$, where $nvar(q) = \{y_0, z_0, y_1, z_1, \ldots, z_{n-1}\}$ are the query's node variables (they need not be distinct), $\mathbf{x} \subseteq nvar(q)$ are the query's head variables, and the r_i, $0 \leqslant i \leqslant n-1$, are regular path expressions. Here, the query has the following properties of branching regular path expressions. [3] For each query conjunct $y_i r_i z_i (0 \leqslant i \leqslant n-1)$, let y_i be the source variable and z_i be the destination variable.

1. Each source variable, except the first one, appears as a destination variable in an earlier step.
2. A variable may appear as a source variable in more than one step.
3. A variable may not appear as a destination variable in more than one step.

Again, for each query conjunct $y_i r_i z_i$, let δ be a function which maps node variables to an infinite set O of oids, i.e. $\delta(U) = o$ where $U \in nvar(q)$ and $o \in O$, there is a path which satisfies the regular path expression r_i between $\delta(y_i)$ and $\delta(z_i)$ in the data graph DB. Each substitution δ defines a tuple in a relation R_q, whose attributes are variables in q. The result of the query q is the projection of R_q on the variables in $\mathbf{x}$.

Example 5. When the query $q(b) : -a(\text{National})b, b(\text{player.nickname.``}BK\text{''})c$ is applied to the database in Fig. 8, the relation $R_q(a, b, c) = \{(1, 2, 7)\}$. Therefore, the result of q is $\pi_b(R_q) = \{2\}$.

4.2.2. Rewriting of multiple regular path queries

As stated earlier, user queries are usually composed of multiple regular path expressions. So, it is necessary to calculate rewritings of multiple regular path queries. The problem is, given the following form of query q, and a set of views v, finding of the query q' which accesses at least one view of v and returns the same result as q.

$$\begin{aligned} &q(\mathbf{u}) : -p_0\ r_0\ p_1, p_1\ r_1\ p_2, p_1\ r_2\ p_3, p_3\ r_3\ p_4, p_3\ r_4\ p_5 \\ &v(\mathbf{w}) : -q_0\ r_5\ q_1, q_0\ r_6\ q_2 \end{aligned} \tag{1}$$

The problem is addressed in [10,31]. Authors in [31] presented an algorithm of query rewriting for TSL, a semistructured query language and showed its soundness and completeness. TSL has restructuring capabilities but does not support regular path expressions. On the other hand, the work in [10] supports regular path expressions but does not support restructuring capabilities. Here, we will show the key idea of the technique in [10]. If a query q and a view v are composed of n and m query conjuncts respectively, there are n^m possible mappings. The technique in [10] improves this potential complexity by introducing the query and view graph.

[3] This is similar to the branching path expression in [26].

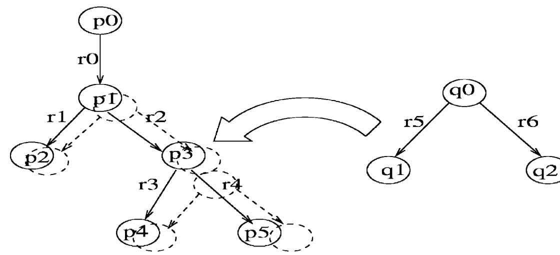


Fig. 7. A query graph and a view graph.

For example, there are 5^2 mappings from the view's body to the query's body in Formula (3). However, Fig. 7 shows that there are only four possible mappings. [4] After applying the symbol mapping, the query has the following form.

$$q'(\mathbf{u}) : -w(\mathbf{x}, \mathbf{y}), c(\mathbf{y}, \mathbf{z}) \tag{2}$$

Here, $w(\mathbf{x}, \mathbf{y})$ denotes rewritten queries using views among query conjuncts and $c(\mathbf{y}, \mathbf{z})$ denotes unchanged query conjuncts. When $c(\mathbf{y}, \mathbf{z}) = \phi$, complete rewritings [24] exist and when $w(\mathbf{x}, \mathbf{y}) = \phi$, there is no rewriting using views. For the rewritten queries, the result of the query is obtained by joining views and each query conjunct, which is unchanged. The basic query evaluation technique is the 2^*-index. It is an expansion of the 2-index [27] which is an index structure for efficiently finding all pairs of nodes that are connected by a regular path expression.

5. XML query processing

In this section, we address efficient XML query processing techniques. We classify the techniques into two categories. The first category is the schema extraction technique and the second category is the graph pruning technique.

5.1. Schema extraction

Query evaluation techniques based on graph traversal are usually inefficient compared to those of traditional database systems since there is no schema fixed in advance. That is, to process given queries targeted to specific schemas in traditional database systems, a query processor can only process the schemas targeted. On the other hand, in semistructured data models, the entire data graph should be processed by a query processor. So, optimization techniques for queries by schema extraction are proposed. We classify the techniques into two groups: techniques of using automata and those of using simulations.

5.1.1. Schema extraction using automata

A schema extraction technique using automata theory is proposed in [19,29]. DataGuides introduced in [19,29] are structural summaries of semistructured databases. The technique regards a

[4] Fig. 7 shows only two mappings but there are four possible mappings as the order of child mappings can be changed.

source database as a non-deterministic finite automaton and creates a data guide that is the corresponding deterministic finite automaton. Formally, the NFA $(Q, \sum, \delta, q_o, F)$ corresponding to an object o in OEM is constructed as follows.

- $Q = state(D) \cup \{end\}$
- $\sum = L \cup \{\perp\}$
- $\delta(state(c), l) = state(object(id))$ *for* $\forall_c \in C$ *and* $\forall <l, id> \in value(c)$
- $\delta(state(a), \perp) = end$ *for* $\forall_a \in A$
- $q_0 = state(o)$
- $F = Q.$

Here, the function *state* maps every object within o to a unique automaton state corresponding to it and maps a set of objects within o to the set of the automaton states corresponding to them. A denotes the set of all atomic objects within o, C the set of all complex objects within o, D the set of all objects within o, and L the set of all different labels of object references within o.

For example, assume that a data graph which represents a partial MLB (Major League Baseball) database. If we regard nodes in the graph as states in an automaton and edges as transitions, Fig. 8 becomes a non-deterministic automaton. That is, for example, there are two transitions labeled 'National' out of state 1, one going to state 2 and another going to state 14. Fig. 9 shows a DataGuide for Fig. 8. Since a DataGuide is a deterministic finite state automaton, its size is usually small compared to the size of the source database. So, we can speed up query processing for semistructured queries by a DataGuide, since it serves as a path index.

For example, consider the regular path expression 'MLB.National.player.nickname'. Without a DataGuide, a query processor would be forced to examine each National, in turn each player of each National and finally return each nickname object of each player object. The bold lines in Fig. 8 shows this process. In the DataGuide technique, any object in a DataGuide graph has a target set that denotes all objects reachable by a given label path in the source database. For example, the target set of the object 2 in Fig. 9 is $\{2, 14\}$ in Fig. 8. In the above example, the query result is the objects in the target set of 'MLB.National.player.nickname'. So, the target set of the object 6 in Fig. 9, that is, $\{7, 18\}$, is returned. This traversing is shown in the bold lines in Fig. 9. In this way, the object search space is reduced.

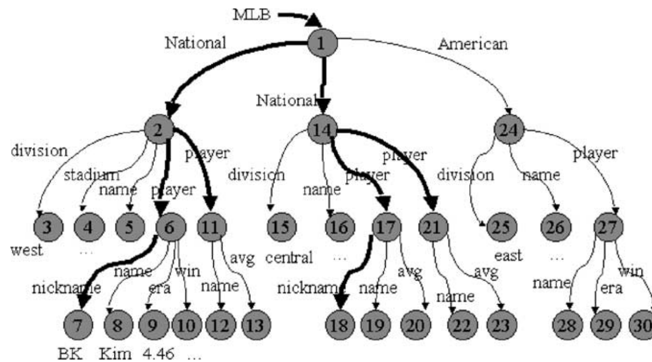


Fig. 8. An example MLB data graph.

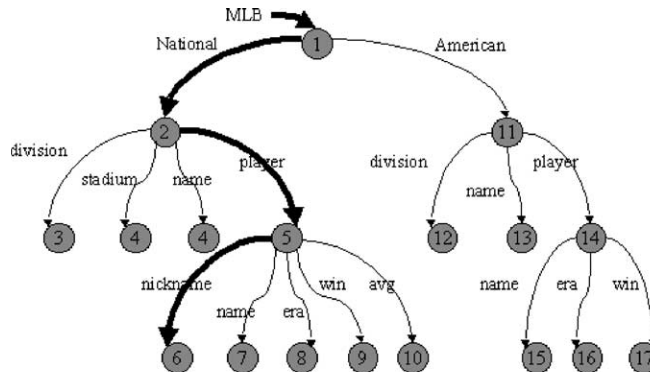


Fig. 9. A DataGuide.

However, this technique can be applied only to queries with a single regular expression. That is, it cannot be directly applied to complex queries with several regular expressions and variables. For example, let's consider the following query which is composed of multiple regular path expressions.

$$q'(b) : -a \ (National) \ b, \ b \ (player.name.\text{“}Park\text{”}) \ c \quad (3)$$

The query asks for teams which have the player named "Park". The result of the query cannot be returned by only traversing the DataGuide graph. That is, if we traverse the DataGuide graph, the results of the query are nodes 2 and 7 and the corresponding target sets are {2,14} and $\{8, 12, 19, 22\}$. However, since there is no path information between nodes 2 and 7, we cannot answer the query by using only the DataGuide graph. Thus, we should traverse the source data graph and the DataGuide graph simultaneously.

5.1.2. Schema extraction using simulation

By constructing a schema graph to which a data graph conforms, we can reduce the search space in query processing and can create a schema graph by using the concept of simulation. If there is a simulation relation between two graphs G_1 and G_2, every edge in G_1 has a corresponding edge in G_2. Formally, given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a relation R on V_1, V_2 is a simulation if it satisfies

$$\forall l \in L \ \forall x_1, y_1 \in V_1 \ \forall x_2 \in V_2 (x_1[l]y_1 \wedge x_1 R x_2 \Rightarrow \exists y_2 \in V_2 (y_1 R y_2 \wedge x_2[l]y_2)) \quad (4)$$

Here, L is a set of edge labels and $[l]$ is a binary relation on $V = V_1 \cup V_2$. The notation $x[l]y$ means that there is an l-labeled edge from x to y. A simulation between a semistructured data instance and a schema graph can be defined by expanding the definition of simulation, as follows.

- The roots of the OEM data and schema graph must be in the simulation.
- An edge $x_1[l]y_1$ in the data graph can be simulated by some edge $x_2[l']y_2$ when l' is a wild card '-' or an alternation containing the label l.

Fig. 10 shows a schema graph corresponding to the data graph in Fig. 8 and Table 2 shows the simulation relation between nodes of Figs. 8 and 10.

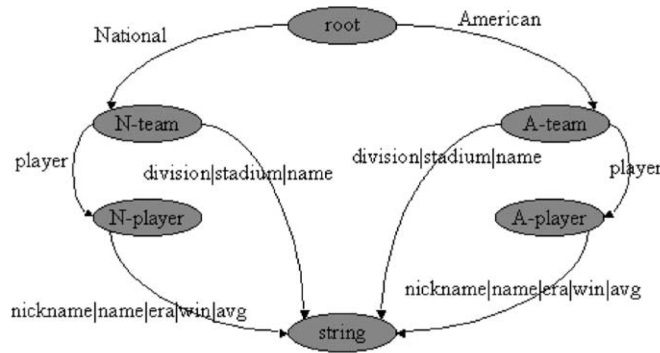


Fig. 10. A schema graph.

Table 2
Data nodes and schema nodes

Data node	Schema node
1	Root
2, 14	N-team
24	A-team
6, 11, 17, 21	N-player
27	A-player
3, 4, 5, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 20, 22, 23, 25, 26, 28, 29, 30	String

Since the size of a schema graph is usually small compared to a data graph, we can reduce the search space in query processing by traversing schema graphs instead of data graphs.

5.2. Graph pruning

Graph pruning techniques which prune the search space of query processing by adding some information to data graphs have been proposed [11,32].

The techniques do not require much additional storage for indexes, and since the structure of the source database to which queries are processed is preserved, they can process complex queries such as those that have more than one regular expressions.

We will address the key idea of the NodeInfo technique [11] as an example. The NodeInfo technique extracts information from DTDs statically and provides a query processor with it at run time. For example, consider the following DTD definition for the element person.

```
<!ELEMENT person (name, address, e-mail*,(school|company))>
```

From the DTD, we can classify the person element into four groups: (1) ones who have one or more e-mail addresses and work for companies, (2) ones who have no e-mail address and work for companies, (3) ones who have one or more e-mail addresses and are students and (4) ones who have no e-mail address and are students. That is, the element person is divided into four groups according to its labels as in Table 3.

Table 3
A classification table

1	{e-mail, school}
2	{e-mail, company}
3	{school}
4	{company}

Table 4
A comparison of query evaluation techniques

	Construction cost	Index size	Single	Multiple
Based on automata	Exponential	Exponential	O	X
Based on simulation	$m \log(n)$	Linear	O	X
Graph pruning (NodeInfo)	$O(kn)$	$O(tn)$	O	O

When each element is classified in this way, the search space can be reduced. For example, when the query that is related to students who have e-mails is processed, the nodes denoting persons who have no e-mail and work for companies need not be traversed.

The NodeInfo technique gives classification information about each object to a query processor. For example, the object &1 in Fig. 1 belongs to 2:{e-mail, company} and the object &3 to 2:{school}. The variable node_info in the NodeInfo technique stores the index of the label set which the corresponding object belongs to in the classification table. For example, the object &1 has node_info of 2 which is an index of a label set {e-mail, company}. So, when a query which is related to company employees who have e-mail address is processed, the node &3 in Fig. 1 which denotes a student need not be traversed.

However, the technique suffers from combinatorial explosion. That is, each occurrence of '*' or '|' doubles the number of classes.

5.3. Comparison

Table 4 shows a comparison of XML query processing techniques. [5] The construction cost and index size is a worst case scenario. The schema extraction using automata shows exponential cost in construction and index size since conversion of an NFA to a DFA requires in worst case time (and space) exponential in the size of the data graph. Schema extraction techniques cannot be directly applied to multiple regular path queries since the structure of the database to which queries are processed is not preserved.

6. Conclusion

Recently, as XML is emerging as the standard for information exchange on the World Wide Web, the problem of XML query evaluation raises significant challenges among database

[5] Assume that the data graph has n nodes and m edges and k and t denote some constants in each algorithm.

communities. As XML queries are composed of arbitrary regular expressions and wild-cards, they usually require large search space during query processing. Thus, many techniques of querying XML documents have been proposed.

As this survey has shown, there are three dimensions in storing and querying XML documents: XML instance storage, XML query languages and XML views and XML query language processing. First, there are two approaches to storing XML documents. One is using special purpose query engines for XML data and the other is using traditional database systems.

In the context of using traditional database systems, the main issue is how to infer database schemas to store XML documents. As described in this article, we classify the techniques in two groups. The first one is storing the graph itself to which XML data is mapped and the second one is inferring schemas from XML DTDs. In this case, the challenge is how to store and query XML documents that have different data models compared to traditional relational or object-oriented data models.

Second, in the dimension of XML query language and XML views we classify techniques based on whether the queries and views are composed of single or multiple regular path expressions.

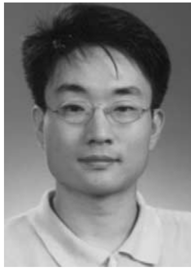
Finally, in the context of XML query processing, the key point is how to reduce the search space of the graph traversal. For this goal, many algorithms including schema extraction, graph pruning have been proposed. In this case, whether the proposed techniques can be applied to queries having multiple regular path expressions and the overhead for enhanced techniques are comparison criteria.

We believe that there are still many challenges that remain open. That is, from theoretical foundations to considerations of a more practical nature, how to process queries having arbitrary regular expressions and to infer appropriate schemas to store XML documents should be addressed.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, The lorel query language for semistructured data, International Journal on Digital Libraries (1996).
[2] Software AG, 2002. Available from <http://www.softwareag.com/tamino>.
[3] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Simeon, XML Path Language (XPath) 2.0. Technical report, W3C Working Draft, December 2001.
[4] T. Bray, J. Paoli, C. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, W3C Recommendation, 1998.
[5] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, A query language and optimization techniques for unstructured data, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1996.
[6] D. Calvanese, G. De Giacomo, M. Lenzerini, M.Y. Vardi, Rewriting of regular expressions and regular path queries, in: Proceedings of ACM Symposium on Principles of Database Systems, 1999.
[7] R.G.G. Cattell, The object database standard: ODMG-93, Morgan Kaufmann Publishers, 1994.
[8] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, M. Stefanescu, XQuery: a query language for XML, Technical report, W3C Working Draft, February 2001.
[9] V. Christophides, S. Abiteboul, S. Cluet, M. Scholl, From structured documents to novel query facilities, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1994.
[10] T.-S. Chung, H.-J. Kim, A two phase optimization technique for XML queries with multiple regular path expressions, Technical report, Seoul National University, 2001.

[11] T.-S. Chung, H.-J. Kim, Extracting indexing information from XML DTDs, Information Processing Letters 81 (2) (2002) 97–103.
[12] T.-S. Chung, S. Park, S.-Y. Han, H.-J. Kim, Extracting object-oriented schemas from XML DTDs using inheritance, in: 2nd International Conference on Electronic Commerce and Web Technologies(EC-Web) with LNCS, 2001.
[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, Query language for XML, in: Proceedings of Eighth International World Wide Web Conference, 1999.
[14] A. Deutsch, M. Fernandez, D. Suciu, Storing semistructed data with STORED, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1999.
[15] excelon corporation, 2002. Available from <http://odi.com>.
[16] M. Fernandez, D. Suciu, Optimizing regular path expressions using graph schemas, in: IEEE International Conference on Data Engineering, 1998.
[17] D. Florescu, D. Kossmann, Storing and querying XML data using an RDBMS, IEEE Data Engineering Bulletin 1 (1999) 1.
[18] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, K. Shim, XTRACT: a system for extracting document type descriptors from XML documents, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2000.
[19] R. Goldman, J. Widom, DataGuides: enabling query formulation and optimization in semistructured databases, in: Proceedings of the Conference on Very Large Data Bases, 1997.
[20] T.J. Green, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata, in: Proceedings of the International Conference on Database Theory, 2003.
[21] T. Grust, Accelerating XPath location steps, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 2002.
[22] Z.G. Ives, A.Y. Levy, D.S. Weld, Efficient evaluation of regular path expressions on streaming xml data, Technical report, University of Washington, 2002.
[23] M. Kifer, W. Kim, Y. Sagiv, Querying object-oriented databases, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1992.
[24] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, D. Srivastava, Answering queries using views, in: Proceedings of ACM Symposium on Principles of Database Systems, 1995.
[25] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, Lore: a database management system for semistructured data, SIGMOD Record (1997).
[26] J. McHugh, J. Widom, Optimizing branching path expressions, Technical report, Stanford University Database Group, 1999.
[27] T. Milo, D. Suciu, Index structures for path expressions, in: Proceedings of the International Conference on Database Theory, 1999.
[28] M. Murata, Extended path expressions for XML, in: Proceedings of ACM Symposium on Principles of Database Systems, 2001.
[29] S. Nestorov, J. Ullman, J. Wiener, S. Chawathe, Representative objects: concise representations of semistructured, hierarchical data, in: IEEE International Conference on Data Engineering, 1997.
[30] Y. Papakonstantinou, S. Abiteboul, Object fusion in mediator systems, in: Proceedings of the Conference on Very Large Data Bases, 1996.
[31] Y. Papakonstantinou, V.A. Vassalos, Query rewriting using semistructured views, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1999.
[32] S. Park, H.-J. Kim, SigDAQ: an enhanced XML query optimization technique, Journal of Systems and Software 61 (2) (2002) 91–103.
[33] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, in: Proceedings of the Conference on Very Large Data Bases, 1999.
[34] D. Suciu, Semistructured data and XML, in: Proceedings of International Conference on Foundations of Data Organization, 1998.
[35] D. Suciu, M. Fernandez, S. Davidson, P. Buneman, Adding structure to unstructured data, in: Proceedings of the International Conference on Database Theory, 1997.



Tae-Sun Chung received his BS degree from KAIST (Korea Advanced Institute of Science and Technology), in 1995 and his MS and Ph.D. degree in school of computer science and engineering from Seoul National University, in 1997 and 2002, respectively. He is currently enrolled in Samsung Electronics Co., Ltd. His research interests include semistructured and XML databases, object-oriented databases, and database programming languages.

Hyoung-Joo Kim received his BS degree in computer engineering from Seoul National University, Seoul, Korea, in 1982 and his MS and Ph.D. in computer engineering from University of Texas at Austin in 1985 and 1988, respectively. He was an assistant professor of Georgia Institute of Technology, and is currently a processor in the Department of Computer Engineering at Seoul National University. His research interests include object-oriented databases, multimedia databases, HCI, and computer-aided software engineering.