Relational-Style XML Query

Taro L. Saito
leo@cb.k.u-tokyo.ac.jp

Shinichi Morishita
moris@cb.k.u-tokyo.ac.jp

Department of Computational Biology
University of Tokyo, Japan
Japan Science and Technology Agency (JST)

ABSTRACT

We study the problem of querying relational data embedded in XML. Relational data can be represented by various tree structures in XML. However, current XML query methods, such as XPath and XQuery, demand explicit path expressions, and thus it is quite difficult for users to produce correct XML queries in the presence of structural variations.

To solve this problem, we introduce a novel query method that automatically discovers various XML structures derived from relational data. A challenge in implementing our method is to reduce the cost of enumerating all possible tree structures that match the query. We show that the notion of functional dependencies has an important role in generating efficient query schedules that avoid irrelevant tree structures.

Our proposed method, the *relational-style XML query*, has several advantages over traditional XML data management. These include removing the burden of designing strict tree-pattern schemas, enhancing the descriptions of relational data with XML's rich semantics, and taking advantage of schema evolution capability of XML. In addition, the independence of query statements from the underlying XML structure is advantageous for integrating XML data from several sources. We present extensive experimental results that confirm the scalability and tolerance of our query method for various sizes of XML data containing structural variations.

Categories and Subject Descriptors:
H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Design, Management

1. INTRODUCTION

XML (eXtensible Markup Language) [6] is a text format for tree-structured data. While it is suitable for describing any type of data, there is no such common data format for relational databases. Hence, XML is a promising portable format for relational data. However, there is no obvious simple manner for making queries of relational data embedded in tree-structured XML.

With regard to the expressibility of data, there is no significant difference between XML and relational data [16]. For example, node and edge tables are sufficient to describe tree-structured data in relational databases. Even so, the tree structure of XML is necessary in several cases. XHTML [9], which is an XML version of HTML, uses a tree structure for data layout, which would not work in relational form. Another case is the use of user-defined tags in XML for organizing data groups or appending additional information to extend the schema of XML dynamically.

Other than these two cases involving data layout and the schema-evolution facilities of XML, various types of data can be expressed in relational format. Hierarchical data, often mentioned as an ideal XML application, are not difficult to describe in relational form using simulated data hierarchies with keys to expand multiple columns. An example of this is shown in **Figure 1**, illustrating relational and XML data with corresponding hierarchies. A triplet of company, section and employee (IDs) comprises a primary key in the following relational data:

company	section	employee
c1	s1	e1
c1	s1	e2
c1	s2	e3

```
<company id="c1">
  <section id="s1">
    <employee id="e1"/>
    <employee id="e2"/>
  </section>
  <section id="s2">
    <employee id="e3"/>
  </section>
</company>
```

Figure 1: Hierarchical data in relational and XML format

This translation from relational data to XML is quite natural and straightforward for the hierarchy of companies through to sections and employees. However, by changing the viewpoints of this relational data, other XML representations are also possible. In **Figure 2**, the XML data on the left-hand side organize the above relational data for each section, and those on the right-hand side are for each employee:

```
<sectionList>
  <section id="s1">
    <company id="c1"/>
    <employee id="e1"/>
    <employee id="e2"/>
  </section>
  <section id="s2">
    <company id="c1"/>
    <employee id="e3"/>
  </section>
</sectionList>
```

```
<employeeList>
  <employee id="e1">
    <company id="c1"/>
    <section id="s1"/>
  </employee>
  <employee id="e2">
    <company id="c1"/>
    <section id="s1"/>
  </employee>
  <employee id="e3">
    <company id="c1"/>
    <section id="s2"/>
  </employee>
</employeeList>
```

Figure 2: Various XML representations of relational data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

Although the meaning of these XML data is the same, XML queries using path expressions are dependent on the specific XML structures. For example, an XPath [8] query to retrieve all employees in a company $c1$ and section $s1$ is completely different for each XML dataset, as shown below:

$P1$: //company[@id='c1']/section[@id='s1']/employee (Fig. 1)
$P2$: //section[@id='s1'][company[@id='c1']]/employee (lhs of Fig. 2)
$P3$: //employee[company[@id='c1']][section[@id='s1']] (rhs of Fig. 2),

where the descendant-axis (//) traverses an arbitrary-depth of XML data, while the child-axis (/) for child nodes, the attribute-axis (@) for attribute nodes (data contained in start tags), and the brackets([ ]) enclose twig nodes to test. This example indicates that without knowledge of the precise XML structure, users cannot produce correct XML queries.

This problem of the *structural variations* of XML data is common when translating relational data into XML. One possible solution to this problem is to disallow structural variations using an XML schema [25], DTD [6], or RelaxNG [18]. However this greatly limits the flexibility of XML data modeling, and prevents dynamic schema evolution or the population of XML nodes with user-defined tags. The requirement for XML schemas comes mainly from the existing standard XML processing methods (e.g., SAX [20], DOM [6], XPath [8], XQuery [5], etc.). These query methods are based on tree navigation, so without detailed knowledge of the underlying XML structure, it is quite difficult to traverse tree-structured XML data correctly.

A brute-force solution would be to cover all structural variations with a single XPath expression by exhaustively concatenating all possible tree patterns. For the above example, the query would be $P1$ | $P2$ | $P3$. However, a slight change in the XML structure, for example when some employees join a project team, and thus XML data are modified as in **Figure 3**, would still force the user to modify query statements or XML reader programs to accommodate this new structure:

```
<company id="c1">
  <section id="s1">
    <team project="p1">
      <employee id="e1"/>
      <employee id="e2"/>
    </team>
  </section>
  <section id="s2">
    <employee id="e3"/>
  </section>
</company>
```

Figure 3: Decorating employee data with a custom tag, team.

Unlike the examples of the above XPath queries, SQL query statements are stable after this sort of schema evolution. For example, the following SQL select statement,

```
SELECT company, section, employee FROM ...
```

can be used without any modification, because a relation consisting of company, section and employee nodes still holds after insertion of the team node.

This observation motivated us to develop a means of querying XML data in *relational style*. For example, we use a simple expression (company, section, employee) to specify node names in a relation without reference to the tree structure, and retrieve variously structured relational data embedded in XML. A key insight in this development is that even if XML representations vary according to the specific viewpoint of relational data, these XML structures are all derived from the same relational data. To describe these variously structured relational data with a simple expression, we define a class of tree structures that construct *relations* in XML. Given a query expression, e.g., (company, section, employee), our query method covers all possible tree structures that can be generated from input company, section and employee nodes.

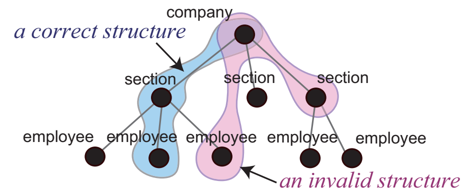


Figure 4: An example of an inappropriate query result.

A challenge in implementing our query method is to discover the appropriate tree structures from the XML data. In general, the number of possible structural variations of n XML nodes is n^{n-1}, which is identical to the number of labeled trees with n nodes. To improve query performance, we must avoid issuing n^{n-1} queries. Another challenge is that even for a single tree pattern, its instances in XML data could be numerous. For example, XML data in **Figure 4** has a hierarchical pattern with one company node, three section nodes and five employee nodes. While there are $1\times3\times5 = 15$ instances of (company, section, employee) pairs, only 5 of those are appropriate in that they connect each employee node with its corresponding parent section node. This shows that naive enumeration of tree instances is inefficient for larger volumes of XML data. Therefore, eliminating incorrect tree structures is another key to achieving good query performance.

To remove irrelevant tree structures from query results, it is necessary to know the implied semantics in the XML data, e.g., each employee node belongs to a section node. We describe these semantics with functional dependencies (FDs) [17] tailored to XML. For example, an FD could be employee $\rightarrow$ section, meaning that each employee node belongs to a unique section node. Our definition of FD is flexible to allow structural variations, as a section node may be a child of an employee node (**Figure 2**), or there may be another node inserted between them, as shown in **Figure 3**.

Our proposed method, the *relational-style XML query*, provides new insight into XML query processing. While the de facto standards for XML query processing languages, such as XPath [8] and XQuery [5], require explicit path expressions to perform queries, we use FDs to define XML data structures, and thus have no need to specify tree structures in query statements. All we need to query XML data is to describe target nodes of interest with tag names, predicates, keywords, etc. Relational-style XML queries enable the user to perform queries without detailed knowledge of the XML structure. This means query expressions are much simpler than those for path-based query methods.

The outline and contributions of this paper are as follows:

- We present a compelling example of the relational-style XML query, which does not use explicit path structures for either queries or schemas (Section 2).
- We define a *relation in XML* that can capture structural variations in XML data, and present an XML algebra to describe XML queries (Section 3).

- We define FDs for XML, and create a relationship between XML structures and FDs (Section 4).
- We present optimization techniques based on our XML algebra to expedite retrieval of XML structures satisfying FDs (Section 5).
- We present experimental evaluation of our proposed methods to confirm the scalability and tolerance of our proposed method. (Section 6)

We present a survey of related work in Section 7, and conclude this work in Section 8.

2. RELATIONAL-STYLE XML QUERY

Relational-style XML query allows structural variations in XML databases. This capability provides a great impact on XML query processing. For example, by detecting relational-part from existing XML data, we call this a *relation* in XML, query expressions of XML becomes much simpler than path-based query methods. In addition, in creating a XML database from scratch, its schema design becomes straightforward translation from an ER-diagram [17], which is far simpler than defining a comprehensive tree schema. To illustrate these benefits, let us consider an XML database of a company data that has several employees and working projects. **Figure 5** illustrates an ER-diagram of this database:

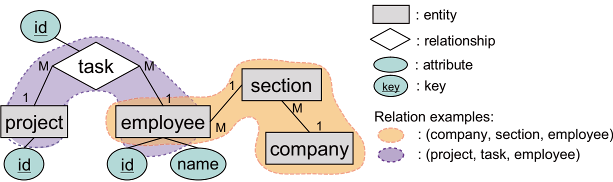


Figure 5: An ER-diagram of a company data, and its decomposition into relations.

To create an XML database from this model, we first decompose this ER-diagram into several relations:

```
R1: (company, section, employee)
R2: (project, task, employee)
R3: (employee, name)
```

We choose these relations so that each of these node pairs organizes a reasonable unit in this data model, so a relation can be a much smaller fragment, e.g., (company, section), (section, employee), etc. This decomposition process is similar to the design of table schemas in relational databases.

One-to-Many Relationship. In this ER-model, a company has several sections, and each employee belongs to one of these sections. This is an example of one-to-many relationships between a company and sections, and a section and employees. To describe these relationships in the ER-diagram, we extract the following functional dependencies (FDs):

employee → section	(Each employee belongs to a section)
section → company	(Each section belongs to a company)

An one-to-many relationship between P and Q corresponds to an FD Q → P, meaning that from each node Q we can uniquely determine another node P. Stated in another way, a node P may have several associated nodes Q.

Many-to-Many Relationship. This data model has a project node, each of them has several tasks. Each task is assigned to an employee, and employees may be assigned several tasks in several projects. This is a many-to-many relationship between projects and employees. In general, we can divide such many-to-many relationships into one-to-many relationships [17]. The following FDs represent two one-to-many relationships (project-task and employee-task):

task → project
task → employee

Relation to XML Structures. Relations and FDs are sufficient to describe a schema of XML. **Figure 6** shows an example of XML data generated from the ER-diagram. This example involves various tree structures that denote data in the same relation. The node pairs of (company, section, employee) are hierarchically organized when ignoring the employee_list node. The tree structures of (project, task, employee) pairs are different under the project_list and task_list nodes. In the traditional XML schema design, we have to decide which structure to use, even though this structural difference has no significant meaning. The relational-style XML query completely does away with the inconvenience, because query expressions for retrieving these distinct tree structures are the same as follows:

```
(project, task, employee)
```

From a given set of definitions of relations and FDs, our query processor automatically finds XML structures that form a relation.

Querying Relational Data Enhanced with XML. XML has rich-data semantics that can enhance the meanings of relational data. For example, a relation (project, task, employee) in **Figure 6** is decorated with an intermediate node, active (17), which does not appear in the ER-diagram. The other nodes employee_list (2), project_list (14) and task_list (26) also enhance relational data by grouping the XML structures representing relations.

In XML, it is required to handle database queries that contain both relational and XML semantics. Consider a query for employee names who are working for active tasks. In **Figure 6**, two task nodes 18, 22 are marked as active, but the ER-diagram has no information of the active node. A query Q1 in **Figure 7**, which is written in XQuery [5], has to traverse several paths, then performs a value-based join operation on employee/@id. To produce this XQuery statement, the user must know that the active node appears only under the project_list node, and employee names are under the employee_list. However, learning such knowledge requires a great deal of efforts and demands the ability to make a complex query.

In the relational-style XML query, this query expression becomes much simpler as shown in Q2 in **Figure 7**, which first retrieves two relations (employee, name) and (active, task, employee), then joins them by using employee@id values. Since we have the knowledge of the FD task → employee, we can avoid invalid node pairs such as (employee (20), active (17), task (22)), which connects irrelevant employee and task nodes. In processing XML queries, we have to correctly extract relations embedded in XML, such as (employee, task), and at the same time to locate XML nodes (e.g., active) associated to these relations.

3. RELATION IN XML

In this section, we define a *relation in XML* that specifies XML structures of interest using a pattern tree, which allows various structure organizations by using the notion of *amoeba* [19]. On this basis, we define an *XML algebra*, which is the foundation for describing XML queries with a nested form of expressions.

Throughout this paper, we use a tree model of XML data, made up of tree nodes with text values and edges. To distinguish element nodes (general tree nodes) and attribute nodes [6], attribute node

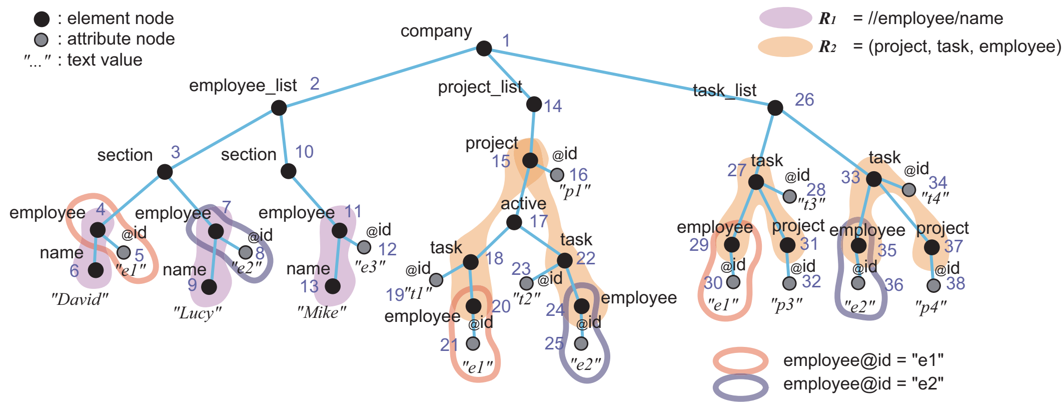


Figure 6: Managing variously structured relations in an XML document.

```
(Q1):  for $x in /company/employee_list/section/employee,
           $y in /company/project_list/project/active/task
       where $x/@id = $y/employee/@id
       return $x/name

(Q2):  (employee, name) join
       (active, task, employee) on employee@id
```

Figure 7: Queries for employee names who are working for active tasks

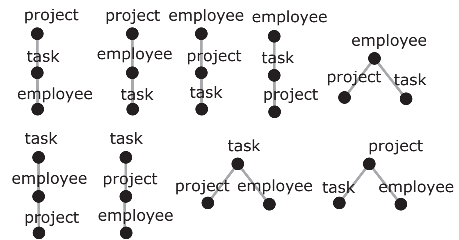


Figure 8: All possible structural variations of project, task and employee nodes.

names are prefixed with "@." Each element and attribute node has a global ID, which is unique in the XML data.

Amoeba Structure. To describe various tree structures that can be generated from XML nodes, the notion of *amoeba* has been proposed [19] as a relaxed definition of trees from the graph theory:

Definition 3.1 [**Amoeba**]. *Given a set $r = \{r_1, \ldots, r_k\}$ of XML nodes, where r_i is an XML node, we say r is an amoeba if one of $r_1, \ldots, r_k$ is a common ancestor of the others, denoted by $\langle\!\langle r_1, \ldots, r_k \rangle\!\rangle$.*

For example, every structural variation in **Figure 8** is an amoeba. To describe a set of amoebas consisting of three types of nodes, project, task and employee, we use a notation ⟪project, task, employee⟫. It is important that regardless of the structure of a node set in the XML data, the node set can be considered to be an amoeba as long as it contains a common root node. The root node of an amoeba is usually an element or attribute node, but a singleton node set, e.g., $r = \{r_1\}$, can also form an amoeba. This definition of amoeba allows node insertions. **Figure 3** shows an example of this where a team node is inserted into the tree structure of company, section and employee nodes.

3.1 Relation in XML

To describe a set of XML data fragments all of which match a specific tree pattern, we need a pattern expression, such as XPath [8]. Such an XPath expression is typically modeled as a pattern tree [11]. However, in the presence of structural variations, it is too restrictive to demand that data structures obey a single tree pattern. Furthermore, concatenating all possible path structures into a single XPath expression can be tedious. To represent both strict and flexible tree structures easily, we introduce the notion of a *relation in XML*, which can express various path structures, including twigs and amoebas:

Definition 3.2 [**Relation in XML**]. *A relation R in XML is a k-ary tuple of nodes (for element and attribute nodes) with a Boolean conjunction of conditions of the following types:*

- *A condition to specify a subset of nodes in R, say $\{a, b, c\}$, constructs an amoeba, denoted $\langle\!\langle a, b, c \rangle\!\rangle$.*
- *For two XML nodes u and $v \in R$, u is a child (or descendant) of v.*
- *Comparison of a text value of a node in R with a constant using one of the operators $=, >, <, \leq$ or $\geq$.*

Although it is possible to use other types of conditions (e.g. document orders of nodes, sibling axes in XQuery [8], etc.), we limit the condition types in a relation for the purpose of illustration.

Definition 3.3 [**An instance of a relation**]. *An instance of a relation R in an XML data, denoted $[\![R]\!]$, is a set of node tuples $\{(r_1, \ldots, r_k)\}$ such that each XML node r_i matches a corresponding node name in R, and satisfies all conditions in R. We denote a node tuple r contained in an instance of R as $r = (r_1, \ldots, r_k) \in [\![R]\!]$.*

A relation in XML can be used to describe a fixed tree structure, which is common in XPath expressions. For example, by using an XPath expression, we simply denote a relation in XML as R_1 = //employee/name to specify a tree pattern consisting of employee and name nodes, where name nodes must be a child of a employee node in the XML document. We denote an instance of R_1 as $[\![R_1]\!]$ or [[//company/name]]. **Figure 9** shows another example of a relation in XML that has element nodes project, task and employee, and a text value [employee@id]. Its predicates are ⟪project, task, employee⟫ and a path constraint that an employee@id node is a child of an employee node. Its instance is shown in the table in

Figure 9. Unlike relational databases, which only use value-based tuples, an instance of a relation in XML can have both node ids and text values. Another extreme example is an XML document, which can be represented as ⟦//*⟧, containing every node in the document.

To denote a relation, consisting of one or more relations $R_1, \ldots, R_k$, we use their Cartesian product $R_1 \times \cdots \times R_k$. For example, if we have R_1 = //employee_list and R_2 = //employee/name, their instances for the XML data in **Figure 6** are $[\![R_1]\!] = \{(2)\}$ and $[\![R_2]\!] = \{(4, 6), (7, 9), (11, 13)\}$. Consequently, the instance of their Cartesian product $R_1 \times R_2$ is:

$$[\![R_1 \times R_2]\!] = \{(2, 4, 6), (2, 7, 9), (2, 11, 13)\}.$$

3.2 XML Algebra

We present three essential algebraic operations for XML queries: *selection*, *projection* and *amoeba join*.

Selection. First, we introduce the selection operation for XML:

Definition 3.4 [**Selection**]. *Let R be a relation in XML, and C be a Boolean conjunction of conditions listed in Definition 3.2. A selection operator, denoted by $\sigma_C(R)$, applies a condition C to a relation R, i.e.,*

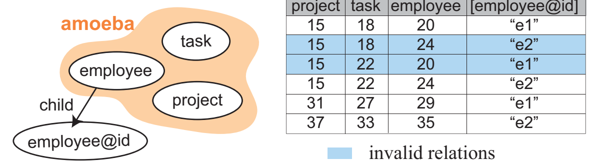
$$[\![\sigma_C(R)]\!] = \{r \mid r \in [\![R]\!] \land r \text{ satisfies } C\}.$$

Node Labels. It is essential to have the capability of specifying some nodes in a relation in XML. In relational databases, a table has columns and each column has a name. Hence, users of the relational database can perform algebraic operations by specifying data columns by name. Node names can be used as equivalents in XML. For example, in an XML relation R_1 =⟪project, task, employee⟫, node names project, task and employee can be used to specify nodes in R_1. To avoid ambiguity of node names between several relations in XML, we use a dot notation. For example, when R_2 = //task/@id and R_3 = //employee/@id, we can distinguish these two @id nodes as R_2.@id and R_3.@id, or we simply denote these node labels as task@id and employee@id. We use a label for a text value of a node n as $[n]$. For example, [task@id] and [name] specify text values for task@id and name, respectively.

In this paper, we consider that the inputs and outputs of an XML query are *relations* in XML, and that a query is evaluated using *instances* of each input relation. Then, the query produces an instance of another relation. In particular, XML queries often involve intermediate results, which are themselves relations. Assigning new temporary node names to all intermediate relations can be a daunting task. Therefore, for readability, we assume node names are *inherited* by the intermediate relations. For example, if we perform a selection operation on a relation R =//book/@isbn and generate another relation R' = //book[@isbn="xx1"], then we can use the node names book and book@isbn that exist in both the relations R and R'.

Projection. To retrieve a specific set of XML nodes from a relation, we define the *projection* of a relation R, denoted by $\pi_{NL}(R)$, where NL is a list of node labels. For example, when R = //employee/name and $[\![R]\!] = \{(4, 6), (7, 9), (11, 13)\}$, then the result of a projection $[\![\pi_{\text{name}}(R)]\!]$ is $\{(6), (9), (13)\}$.

Amoeba Join. Given a list of relations in XML, R_1 = //project, R_2 = //task and R_3 = //employee, for example, we need an operation to construct their amoebas. This operation is called an *amoeba join* [19]. A similar operation is a *structural join* [1], which concatenates two nodes p and q if p is an ancestor of q. The structural join is generally used to process descendant-axis (//) queries. However,



project	task	employee	[employee@id]
15	18	20	"e1"
15	18	24	"e2"
15	22	20	"e1"
15	22	24	"e2"
31	27	29	"e1"
37	33	35	"e2"

Figure 9: An example of an relation in XML (left) and its instance (right) in Figure 6. The colored rows are invalid instances violating FDs (see Section 4).

to handle structural variation, we also must consider both the case where p is an ancestor of q, *or* p is a descendant of q. In addition, there are indirect structural relationships involving more than two nodes, for example, nodes p and q connected through another node r. To collect instances of variously structured XML data, we describe the amoeba join operation as an operator in the XML algebra:

Definition 3.5 [**Amoeba Join**]. *Given a list of node labels L_1, ..., L_m, and a list of input relations $R_1, \ldots, R_k$, an amoeba join operation $AJ_{L_1,\ldots,L_m}(R_1, \ldots, R_k)$ is a selection with an amoeba condition for $L_1, \ldots, L_k$, i.e.,*

$$AJ_{L_1,\ldots,L_m}(R_1, \ldots, R_k) = \sigma_{\langle\!\langle L_1,\ldots,L_m \rangle\!\rangle}(R_1 \times \cdots \times R_k).$$

For example, when R_1 = //project, R_2 = //task and R_3 = //employee, then an amoeba join $AJ_{\text{project,task,employee}}(R_1, R_2, R_3)$ is a selection with a condition ⟪project, task, employee⟫, and generates all instances of amoebas in the XML document, matching one of the structures in **Figure 8**.

4. FUNCTIONAL DEPENDENCIES

A relation in XML has the capability of handling variously structured XML data. However, without knowledge of the semantics hidden in XML data, it is not possible to retrieve correct XML structures. For example, **Figure 9** shows invalid tuples (colored in blue) that connect irrelevant task and employee nodes (18, 24) and (22, 20). To resolve this problem, we need information of data semantics, such as each task belongs to a project and is assigned to an employee. These data semantics are described with FDs, task → project and task → employee. In this section, to incorporate data semantics into XML, we define FDs in XML and a class of relations that can be used to describe XML structures satisfying FDs.

We describe a functional dependency for XML with node labels in relations. Let X and Y be lists of node labels. Then, a functional dependency for XML is expressed as $X \to Y$. Now, we give the definition of FDs in XML:

Definition 4.1 [**FDs in XML**]. *We say a relation R satisfies an FD $X \to Y$ if for each pair of instances $p, q \in [\![R]\!]$, $p.X = q.X$ implies $p.Y = q.Y$, where $p.X$ denotes a list of nodes (or text values) in p corresponding each node label in X. The equality of two nodes (or text values) n_1, n_2 is defined as follows:*

$n_1.id = n_2.id$ (*when n_1 and n_2 are XML nodes),*
$n_1 = n_2$ (*when n_1 and n_2 are text values),*
where n.id is a unique node ID in the XML data.

Intuitively, an FD $X \to Y$ specifies that a node set belonging to X uniquely determine a node belonging to Y. For example, some instances in **Figure 9** violate the FD task → employee; two distinct employee nodes 20 and 24 are associated to each of the task nodes

18 and 22. These invalid node pairs are involved due to the flexibility of amoeba structures. In the next section, we solve this problem by restricting allowable XML structures in describing relations.

4.1 Tree Relation

In our definition of FDs, any relation consisting of an arbitrary node set can be used, since a relation in XML not always have a tree structure, such as a projection result, etc. However, in describing a relation instance as an XML data, it is convenient that we have a template structure for reading and writing XML data, such as a table row in relational database. As its counterpart in XML, we use amoeba structures that can be embedded in XML data. However, an amoeba structure itself is a connected component of tree nodes, and thus invalid nodes may be connected, as illustrated in **Figure 9**. To avoid these irrelevant node connections, while allowing various tree structures in describing XML data, we introduce a restricted class of XML structures, called a *tree relation*.

Before defining a tree relation, we introduce some notations. Let F be a set of FDs, $NL(F)$ is the set of node labels appearing in F. Given a list I of relations $R_1, \ldots, R_k$, then if each R_i contains at least one node label in $NL(F)$, and all node labels in $NL(F)$ are contained in I, we say that I *covers* $NL(F)$. For example, for F ={employee → name}, then $NL(F)$ ={employee, name}, and thus the pair of relations R_1 = //employee and R_2 = //name covers $NL(F)$.

Now, we define a tree relation in XML:

Definition 4.2 [**Tree Relation**]. *Let F be a set of FDs and $R_1, \ldots, R_k$ be a list of relations that covers $NL(F) = \{L_1, \ldots, L_m\}$. A tree relation R for F is a result of selection $\sigma_C(R_1 \times \cdots \times R_k)$ such that R satisfies all FDs in F, and C is a conjunction of the following amoeba conditions:*

(P1)	$\langle\!\langle L_1, \ldots, L_m \rangle\!\rangle$	$L_i \in NL(F)$
(P2)	$\langle\!\langle X, Y_1 \rangle\!\rangle \wedge \cdots \wedge \langle\!\langle X, Y_j \rangle\!\rangle$	*for each FD* $X \to Y_1 \ldots Y_j \in F$

where X is a list of node labels, and each Y_i is a single node label.

For example, when $F = \{A \to B,\ B \to CD\}$, then $NL(F) = \{A, B, C, D\}$, and its tree relation for F has the following condition:

$$\langle\!\langle A, B, C, D \rangle\!\rangle \wedge \langle\!\langle A, B \rangle\!\rangle \wedge \langle\!\langle B, C \rangle\!\rangle \wedge \langle\!\langle B, D \rangle\!\rangle.$$

As another example, an FD with the form $AB \to C$, which has several node labels in the left hand side, imposes the constraint $\langle\!\langle A, B, C \rangle\!\rangle$.

The first constraint $(P1)$ $\langle\!\langle L_1, \ldots, L_m \rangle\!\rangle$ confirms that nodes in $NL(F)$ construct an amoeba, i.e., a node set of $L_1 \ldots, L_m$ must at least form a tree structure in the XML data. The second constraint $(P2)$ indicates that nodes appearing in an FD must also have an amoeba structure. Intuitively, to establish the correspondence between FDs and XML structures, we consider XML nodes that construct an amoeba structure are semantically related. If there are partial dependencies (FDs) within a relation, XML structures must represent all of these relationships. **Figure 10** illustrates variations of tree relations for several sets of FDs; A tree relation of nodes A, B and C must form a tree structure but allows several tree shapes. When FDs are defined in this relation, tree shapes are restricted so that these FDs can be represented in these tree structures.

These structural constraints imposed by FDs have an important role in eliminating incorrect XML structures that do not match the data semantics. For example, when F has two FDs task → project and task → employee, then a tree relation for F must satisfy the following condition:

$$\langle\!\langle \text{project, task, employee} \rangle\!\rangle \wedge \langle\!\langle \text{task, project} \rangle\!\rangle \wedge \langle\!\langle \text{task, employee} \rangle\!\rangle.$$

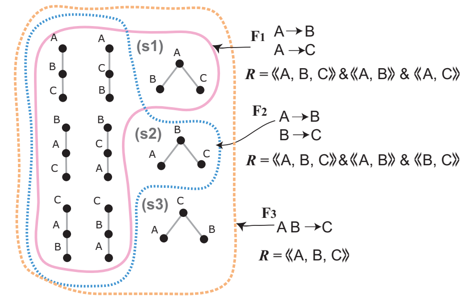


Figure 10: Structures of tree relation (A, B, C) vary according to a set F of FDs.

In **Figure 6**, an instance of a relation R_2 satisfies all of these conditions. Thus, we say R_2 is a tree relation for F. The first constraint 《project, task, employee》 allows all possible tree structures consisting of these three nodes. However, a node pair (15, 18, 24) in **Figure 9** satisfies 《task, project, employee》 but connects irrelevant task (18) and employee (24) nodes. Hence, the other constraints 《task, project》 and 《task, employee》, which are imposed by FDs, are needed to remove such inappropriate tree structures.

Next, we present some examples of FDs in XML:

- employee → employee@id : Each employee node must have an @id attribute node.
- employee@id → employee : The is the opposite of the FD above. In XML, every attribute must belong to a single element, so this type of FD always holds for all attribute nodes.
- author → paper : Each author belongs to a paper. In other words, a paper may have several authors. The rationale to use an amoeba structure 《author, paper》 to represent this one-to-many relationship is that, for each paper node, its author nodes should be ancestor or descendant nodes, not sibling or other nodes. The amoeba condition 《author, paper》 covers such tree structures. If several paper nodes are found for an author node, such XML data violate this FD, and needs to be modified.
- [book@isbn] → book : Given an book@isbn value, we can uniquely determine a book node. In this case, the book@isbn value is a key (global ID) of book node, no duplicate value of book@isbn is allowed in the XML document.
- country, ssn → person : Any person node is identified by a pair of country and ssn (social security number) nodes. This is an example of a primary key with two nodes. Either of the country or ssn nodes is not sufficient to locate a person node, as an ssn may not be unique outside of a country.
- country, [person@ssn] → person : With the information of an country node and person@ssn value, a unique person node can be determined. This example can be considered as a *relative key* [7], which localizes the key definition under the specified path, as uniqueness of [person@ssn] values is also localized in the context of the country node, but various data structures are allowed compared to the relative keys proposed in [7]. For example, a country node can be a parent or child of a person node in our definition of FDs.

5. QUERY PROCESSING

5.1 Pushing Structural Constraints

Using the operations defined so far, we are able to implement the

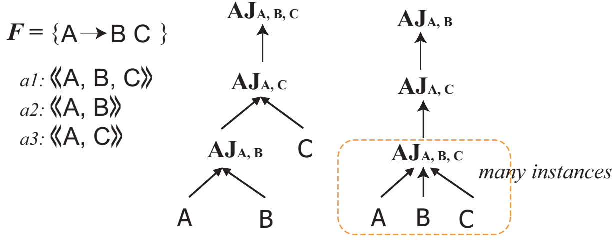


Figure 11: Selection (amoeba join) order affects the performance of the query processing.

"pushing selection" technique developed for relational databases, which makes it efficient to process tree relations for a set of FDs.

Queries for a tree relation for a set of FDs contain several amoeba predicates. As we explained in Section 4, amoeba constraints imposed by FDs eliminate irrelevant structures to the tree relation. The order in which these conditions are applied is an important factor in reducing the size of the intermediate query results.

In this section, we present optimization techniques that translate a query operation into a nested form of several amoeba joins so that temporary results can be minimized by gradually applying structural constraints imposed by FDs. This method enables selective retrieval of XML structures that satisfy each amoeba constraints, and avoids extraction of unwanted XML structures. To give an equivalent translation of a query expression, we incorporate the commutative law and cascading selection of relational algebra [17] into XML:

Theorem 5.1 **[Pushing Selection]**. *Let R and S be input relations, and C be a condition. When a relation S contains no node label that appears in C, the following translations hold:*

$$\sigma_C(R \times S) = \sigma_C(R) \times S \quad \text{(commutative law)}$$
$$\sigma_{c_1 \wedge c_2}(R) = \sigma_{c_1}(\sigma_{c_2}(R)) \quad \text{(cascading selection)},$$

where c_1 and c_2 are conditions.

Proof (Sketch). The proof is an induction on the number of conditions based on the fact that relations of the left-hand side and right-hand side in the above expressions have the same set of conditions. □

Using the rules in Theorem 5.1, we can decompose a selection operation to retrieve a tree relation into a nested form of selections. If A is a set of amoeba conditions, and a is an amoeba condition in A, then the following translation holds:

$$\sigma_A(R \times S) = \sigma_a(\sigma_{A-\{a\}}(R \times S)) = \sigma_a(\sigma_{A-\{a\}}(R) \times S),$$

where a relation S does not contain node labels in $A - \{a\}$.

When $a = \langle\langle X, Y\rangle\rangle$, a selection operation σ_a is an amoeba join $AJ_{X,Y}$ that connects nodes X and Y . Hence, this decomposition technique can be used repeatedly to derive a series of amoeba joins equivalent to the original query. **Figure 11** illustrates query schedules generated by this decomposition. In this example, we have three amoeba conditions, ⟪A, B, C⟫, ⟪A, B⟫ and ⟪A, C⟫. When we choose one of the conditions, say ⟪A, B, C⟫ ($a1$), as a decomposition target, the query schedule becomes like the left-hand schedule in **Figure 11**. This schedule effectively reduces the search space of possible relations by evaluating amoeba conditions $a3$ and $a2$ in earlier steps, thus decreasing the input size of the final $AJ_{A,B,C}$ operation. On the other hand, the right-hand schedule evaluates the condition $a1$ first, which enumerates all possible structural variations, and subsequently makes selections with ⟪A, B⟫ and ⟪A, C⟫.

This translation is equivalent to the so-called *pushing-down* selection in relational algebra. This technique is also useful in XML query processing to eliminate instances of irrelevant XML structures from intermediate query results.

Parent-Child Join Decomposition. Functional dependencies are frequently observed between parent and child nodes, e.g., task@id → task, which imposes ⟪task@id, task⟫, and the task node must be the parent of the task@id node. In this case, we can explicitly decompose the query using a parent-child join:

Corollary 5.2. *Let R and S be input relations, and A be a set of amoeba conditions. For an amoeba condition $a = \langle\langle P, C\rangle\rangle \in A$, where P is the parent node of C, and when a relation S does not contain node labels in $A - \{a\}$, the following translation holds:*

$$\sigma_A(R \times S) = PC_{P,C}(\sigma_{A-\{a\}}(R) \times S),$$

where $PC_{P,C}$ denotes the parent-child join, which is a specialized version of an amoeba join that connects parent nodes P and child nodes C.

5.2 Minimal Relation

Unlike relational databases that use flat tables, relations in XML have tree structures. This structural discrepancy often demands an XML query to involve extra nodes that do not necessarily appear in the final results. For example, consider a query for a pair of project and employee nodes from a relation (project, task, employee). In SQL, simply specifying project and employee labels is sufficient to produce this query statement. In XML, however, we also have to include task node label in the query operation, because when a task node is a root node of the amoeba structure, the project and employee nodes cannot be connected without the task node. Therefore, project, task, customer is a minimal relation required to answer this query.

The algorithm to compute minimal relation for a given list L of input node labels is simple. Let F_q be a subset of pre-defined FDs such that each FD in F_q contains some node label appeared in L. Then, the minimal relation of the node list L is $NL(F_q) \cup L$, which is a list of all node labels that appear in F_q and L. For example, when L = {project, employee} and a pre-defined set F of FDs is {task → project, task → employee}, then F_q is identical to F, and $NL(F_q)$ = {project, task, employee}, which is the minimal relation of (project, employee). Its query operation is described as follows:

$$\pi_{\text{project, employee}}(\sigma_C(\text{project} \times \text{task} \times \text{employee})) \quad (S_1)$$

where a condition C = ⟪project, task, employee⟫ ∧ ⟪task, project⟫ ∧ ⟪task, employee⟫, which is derived from F_q. This query correctly locates the minimal relation for the project and employee nodes, and then the projection eliminates the task node, which is not contained in the original input. The notion of minimal relations can be utilized to complement some missing nodes in the query, so the users can produce queries without considering structural differences between relational and XML structures. For example, a query statement for S_1 is simple as follows:

`(project, employee)` (S_1)

There is a case that some node labels in a query do not appear in any FDs. This is usual for relational data enhanced with XML syntax, as explained in Section 2. For example, a minimal relation of (task_list, task, employee) has no additional node, since an only FD related to this relation is task → employee, but its node labels are already contained in this relation. This query is evaluated with a

nested form of amoeba joins using the query translation technique described in the previous section:

$$AJ_{\text{task_list, task, employee}}(AJ_{\text{task, employee}}(\text{task, employee}), \text{task_list}) \quad (S_2)$$

which first retrieves a relation (task, employee), then finds task_list nodes associated to this relation.

5.3 Database Integration

XML is a tree-structured data, however, a single tree is not sufficient to describe data models of the real world that often should be described as a graph structure. As we illustrated in Section 2, any graph-structured data model can be decomposed into several trees (relations). To integrate several trees into a single XML document, H. V. Jagadish et al. introduced the notion of colorful XML [12], which appends a color property to each XML node so that a projection of each colored tree represents one of the trees decomposed from a graph-structured data. However, the colorful XML requires a significant extension of the XML specification [6], and also to edit multiply colored XML data is quite difficult for standard text editors or simple script programs.

Our solution to this problem is to store several aspects of the data model separately in the form of XML data fragments, and to retrieve them using relational-style XML queries. These query results are joined using *keys* defined for XML. This approach does not require any extension of the XML specification. **Figure 12** illustrates this approach. This XML data has some employee data (left), and associated office and section XML data (right) that wrap employees:

```
<employee id="e1">
  <name>David</name>
</employee>
<employee id="e2">
  <name>Lucy</name>
</employee>
```

```
<office location="L.A.">
  <employee id="e1"/>
  <employee id="e2"/>
</office>

<section id="s1">
  <employee id="e1"/>
  <employee id="e2"/>
</section>
```

Figure 12: Employee data (left) and additional information (office and section) described in two separated trees (right).

These three XML fragments might be placed in the same XML document, or in different XML files. The colorful XML [12] merges these three XML fragments into a single tree while tolerating employee nodes with different colors. This method enables an XML query processor to traverse name, office and section nodes from an employee node. Our solution to this problem is much simpler and leaves the XML data as they are, because a query for employee names, office and section can be expressed as a join operation of relations using employee@id values, described as follows:

$$(\text{employee, name}) \bowtie_{\text{employee@id}} (\text{office, employee}) \bowtie_{\text{employee@id}} (\text{section, employee})$$

Let R, S be relations, and p be a node label for a join target, then a join operation $R \bowtie_p S$ is a selection $\sigma_{R.p=S.p}(R \times S)$. Therefore, without actually materializing a merged form of XML fragments, we can integrate the above XML data from the knowledge that empoloyee@id values connect three relations; namely, employee@id is a *key* (or foreign key) for relations (employee, name), (office, employee), and (section, employee).

A key is a special case of an FD, and it can be used to uniquely locate XML nodes. In this example, we have the following key definitions for these three relations:

$$[\text{employee@id}] \rightarrow \text{employee name} \quad \text{for (employee, name)}$$
$$[\text{employee@id}] \rightarrow \text{office employee} \quad \text{for (office, employee)}$$
$$[\text{employee@id}] \rightarrow \text{section employee} \quad \text{for (section, employee)}$$

These keys (FDs) mean that an employee@id value is sufficient to uniquely locate all nodes in each relation (employee, name), (office, employee) and (section, employee). Buneman et al. have proposed keys for XML [7], however, their definition cannot handle structural variations. Our definition of FD allows both the cases that an office node is a child (descendant) of an employee node, or vice versa.

Integration of variously structured XML data is also useful for handling schema evolution. **Figure 12** illustrates a process of enhancing employee data by appending supplementary information. Suppose that, first, we have only the employee name data, and subsequently these employees are assigned to some office and section, which is described as the right-hand side XML data in **Figure 12**. When creating a new database, it is usual that some data are missing or not available yet. With the capability to query variously structured XML data, schema evolution of XML databases can be managed with a simple join operation of several XML data. In addition, it is flexible to allow various XML structures in designing new XML data for enhancing existing databases.

Related to database integration, we mention several open problems that still need further study:

Handling Variations of Tag Names. There may be variations of tag names in describing the same data model in XML. For example, an XML tag employee may be named worker in another location. To handle these variations of tag names, one can use, for example, a simple mapping function that translates worker into employee or some dictionary that groups synonym words. In general, however, we have to consider a more difficult problem, called *semantic integration* [4], which needs to resolve semantic heterogeneity of XML tag names under specific paths.

Semantics of Nested Elements. When XML data has a recursive structure, its data semantics may be ambiguous. **Figure 13** illustrates this problem; two name nodes are located under the manager node. To query a manager name, the amoeba condition ⟪manager, name⟫ cannot be used, since the manager is associated with its correct name Lucy as well as its employee's name David unexpectedly. A solution to this problem is to clarify the data semantics by using XML namespace e.g., manager:name, employee:name, etc. XML attributes, such as manager@name, also can be used to avoid the problem of the semantic ambiguity. Although it is quite easy to capture the amoeba structure ⟪manger, manager:name⟫, the problem of automatic assignment of these namespace labels remains open.

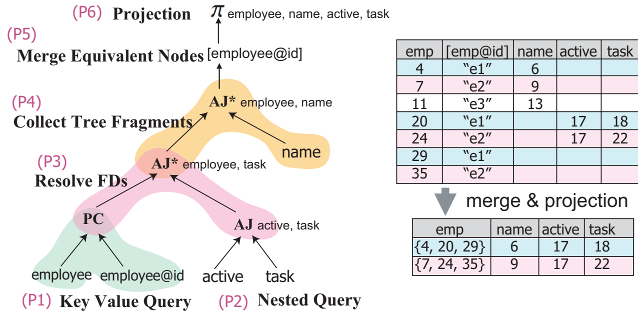
```
<manager>
 <info>
   <name>Lucy</name>
 </info>
 <employee>
   <name>David</name>
 </employee>
</manager>
```

```
<manager>
 <info>
   <manager:name>Lucy</manager:name>
 </info>
 <employee>
   <employee:name>David</employee:name>
 </employee>
</manager>
```

Figure 13: Clarifying semantics of the name tags by using XML namespace.

5.4 Querying Incomplete Relations

Although the relational-style XML query manages structural variations of XML data, the user who only has a limited knowledge of



emp	[emp@id]	name	active	task
4	"e1"	6		
7	"e2"	9		
11	"e3"	13		
20	"e1"		17	18
24	"e2"		17	22
29	"e1"			
35	"e2"			

emp	name	active	task
{4, 20, 29}	6	17	18
{7, 24, 35}	9	17	22

Figure 14: Query schedule of (employee, name, (active, task)), which computes incomplete relations, then merges them to fill blank columns.

the underlying XML structure may fail to retrieve necessary information from the XML data. For example, a query for employee names who are working for active tasks can be described as follows:

(employee, name, (active, task)),

which has a nested query (active, task) to retrieve task nodes marked as active. This query has to find a relation (employee, name, active, task), but there is no matching tree structure for this relation in the XML data in **Figure 6**. In reality, many partially matching structures are available and would provide useful information.

To detect these partial matches, we present a query operation that collects incomplete relations allowing *null* values. For example, the query process of (employee, name, (active, task)) involves node pairs (4, 6, *null*, *null*), (20, *null*, 17, 18) and (29, *null*, *null*, *null*). **Figure 14** shows these nodes tuples. Then, to fill *null* values in these node pairs, we merge employee nodes 4, 20 and 29 by using equality of the employee@id value "e1", and generate a node tuple ($\{4, 20, 29\}$, 6, 17, 18) as one of the query results. In this query process, employee@id values work as object IDs of employee nodes.

We extend the definition of the amoeba join to tolerate *null* values in the query result:

Definition 5.3 [AJ^*]. *Let NL be a list of node labels, and R be an input relation, an amoeba join allowing null values, denoted $AJ^*_{NL}(R)$, generates the same relation with an amoeba join $AJ_{NL}(R)$, except that each result instance in $AJ^*_{NL}(R)$ is allowed to have null nodes other than the node corresponding a first node label in NL.*

The AJ^* operation has a flavor of the *outer join* in relational databases, but is different in that AJ^* considers structural variations of input nodes.

Figure 14 illustrates a query schedule of (employee, name, (active, task)) that uses AJ^* operations instead of AJ. First, to merge employee nodes using employee@id values, this schedule performs PC-join of these nodes (P1). Then, to retrieve task nodes that are marked active, we simply compute their amoeba join (P2). Among the inputs of the query, a pair of employee and task has a structural constraint imposed by the FD task → employee, so we have to connect them by using a AJ^* operation (P3) allowing null values for the task nodes. In the similar manner, we perform AJ^* operation between employee and name to compose a relation (employee, name) (P4). The upper-right table in **Figure 14** shows the intermediate query results up to (P4) phase. In (P5), employee nodes that have the same employee@id values are merged to fill the blank column in the table, and incomplete rows that still have *null* values are eliminated. Finally, using projection π, the query reports only requested nodes by the user, excluding employee@id column (P5), and the result is the lower-right table in **Figure 14**.

5.5 Amoeba Join Processing

The amoeba join processing depends on the capability to detect an ancestor-descendant relationships of two nodes, because to test an amoeba condition $\langle\langle a, b, c\rangle\rangle$, we need to check one of the nodes among a, b and c is a common ancestor of the others. If node a is a common ancestor in the amoeba structure, then the node a is an ancestor of nodes b and c.

To make faster the detection of ancestor-descendant relationships, we use indexes that label each XML node with an interval (start, end) [14]. The tree structure of XML is encoded so that every interval of an ancestor node subsumes all its descendant nodes, and all intervals are disjoint. Using this node label, the detection of the ancestor-descendant relationship becomes a containment test of two intervals, i.e. a node p is an ancestor of a node q iff $p._{start} < q._{start} \wedge q._{end} < p._{end}$.

The details of the amoeba join algorithm are described in [19], thus we present its outline. The amoeba join can be processed efficiently by sorting input nodes in advance in the order of start values, since the root node of an amoeba always has the smallest start value. By sweeping the sorted input nodes, the amoeba join chooses a node p that has the smallest start value as a candidate of the root node of an amoeba. Then, for each input node list of the amoeba join except that contains p, it searches the descendant nodes of p from range between $p._{start}$ and $p._{end}$ for the other components of the amoeba. After the search, this algorithm enumerates all amoeba structures rooted by p, therefore, it sweeps the node p off from the input, then proceed to the next smallest node.

6. EXPERIMENTAL RESULTS

We evaluated the performance of the relational-style XML query to show the scalability of our method for various sizes of XML data, and the tolerance to structural variations.

Implementation. We implemented a prototype of our database management system in C++, which consists of several components, such as XML reader, index generator, query processor, etc. Our implementation of database indexes uses B+-trees provided by the Berkeley DB library [22]. On top of the B+-tree, we stored XML nodes labeled with (start, end, level, path ID, text), where the start and end is the interval labels [14] to efficiently detect ancestor-descendant relationships, and the level is the depth of a node in the XML tree, which is required to detect parent-child relationships of XML nodes. The path ID represents an ID assigned to each distinct path. The text is a text content encapsulated by tags or attributes.

XML nodes are stored in a B+-tree in ascending order of their start values. To make node retrieval faster, we also generated a secondary B+-tree index using a compound key (path ID, start), which aligns XML nodes first in the order of path IDs, then that of start values. This secondary index is used to efficiently locate nodes belonging to specific paths, e.g. //A, //A/B, etc.

Machine Environment. As a test vehicle, we used a Windows XP machine; dual Xeon 3.0GHz processors, 2GB memory and 250GB 7,200 rpm HDD.

Experimental Methodology. We run each query six times and take the average of the last five runs, because OS caches of the database files are quite different between the first run and the others. The standard deviation of the query performance is at most 0.02 (3σ = 0.06 seconds) or a far smaller value. It is sufficiently small to measure differences of the query performance.

Query Performance on XMark. To evaluate the query performance on standard XML data, we used XMark [21] benchmark

program. We have changed its scalability parameter f from 0.1 to 1 to produce various sizes of XML data, which are almost 10M, 25M, 50M and 100M bytes. **Figure 15** shows query schedules used in this experiment (Q_1 to Q_{6S}). This query set is designed so that the characteristics and scalability of the amoeba join algorithm become clear, so simple path queries and join ($\bowtie$) operation that can be processed with the standard techniques are not presented.

The XMark database contains 83 types of tag names. A relation in XML is a subset of these tag names. To detect FDs in the XMark data, we created a simple program that investigates one-to-many or one-to-one relationships that hold in the XMark data. For example, under the root node "site" in the XMark data, there are many person nodes, and each person node has many descendant interest nodes. These relationships correspond to FDs person → site and interest → person.

Query Q_1 and Q_2 are amoeba joins of two nodes that have one-to-many relationships. **Figure 16** shows the performance of these queries and their result sizes. The performance of Q_1 and Q_2 scales in proportion to the XML data sizes.

Here, we present two examples that emphasize the significant benefit of query optimization. When more than two nodes involved in the amoeba join operation (Q_3), its performance significantly deteriorates. Our implementation of the query processor does not use secondary storages to store intermediate results of a query. The permutation size of site, person and interest nodes is quite huge, and consequently the query Q3, which simply computes all possible tree structures consisting of these nodes, exhausted the main memory storage, and stopped after an out of memory error was observed. Query Q_{3F} is an optimized query schedule of Q_3 using the pushing-structural constraint technique described in Section 5, and the amoeba constraints derived from the FDs person → site and person → interest are pushed into the sub queries. Although both Q_3 and Q_{3F} has the same amoeba join operation $AJ_{\text{site, person, site}}$, the performance of Q_{3F} scales well with increase in XML data sizes. This is because nested amoeba join queries in Q_{3F} construct appropriate tree structures in a bottom-up fashion, and efficiently avoids irrelevant tree structures. This result indicates that the right-hand schedule in **Figure 11**, which first processes an amoeba condition with more than two nodes, must be avoided. Query Q_4 and Q_{4F} are more complex examples of nested query schedules. To retrieve the relation (regions, item, mail, date), Q_4 considers an FD mail → date in the path mail/date, so PC-join can be used in this query. However, the relation (regions, item, mail, date) in the XMark data has several other FDs as shown in Q_{4F}. Similar to the results of Q_3 and Q_{3F}, computation of Q_4 could not be completed in the main memory, and Q_{4F}, which considers all of these FDs, is scalable to the database size.

Query Q_5 and Q_{5F} show that amoeba join is not always slow; In XMark data, the mail object is a parent of two child nodes, 'from' and 'date', so the amoeba join of these nodes never reports incorrect results. In this case, the decomposed schedule Q_{5F} is less efficient due to the overhead of pipelining. Query Q_6, Q_{6F} and Q_{6S} retrieve nested relations in which each open_auction node has current price information and several bidders associated with the bid time and amount of increase data. Query Q_6 misses the one-to-many relationship between open_auction and bidder, so Q_{6F}, which totally decomposes the schedule, becomes efficient. Considering that two relations (open_auction, current) and (bidder, increase, time) comprise distinct objects, and are connected through an FD bidder → open_auction, we can produce a more efficient query schedule Q_{6S}, which reduces the number of sub queries. This type of query optimization needs to be exploited but is left as a future work.

Tolerance to Structural Variations. To further study the tolerance of our method for variously structured XML data, we developed an XML data generator that produces three types of structural variations: *simple, hierarchical*, and *random*. **Figure 17** illustrates these tree-structures generated from the same input table. The *simple* structure converts each row in the table into an XML fragment organized from the first column data to the last one. A column value in the input table is described as an XML attribute. The *hierarchical* structure aggregates column values that have the same value. For example, all values in the column a are aggregated into a single tag. This aggregation process is repeated recursively from column a to c. This type of aggregation is frequently observed in the real-world XML data. The *random* structure is generated in almost the same manner with the hierarchical structure, but it randomly chooses target columns of aggregation, so the random XML data contains many structural variations. The generated XML data is a collection of a relation (a, b, c) that satisfies two FDs c → b and b → a, representing two one-to-many relationships. The *fanout* parameter controls the number of associated nodes in these relationships. For example, when fanout = 5, each a node has 5 b nodes, and each b node has 5 c nodes. We programmed this data generator so that all three types of XML data consist of the same number of instances of the relation (a, b, c).

Figure 18 shows the query performance of Q_7 grouped by various query result sizes, and next by fanout values. Even in the presence of structural variations, the query performance between the simple and random format is stable. This characteristic is suited for integrating variously structured XML data. When the fanout parameter is between 2 to 100, the hierarchical data is more efficient for query processing, because it efficiently aggregates one-to-many relationships, and thus its database sizes are smaller than those of the others. However, when the fanout values are 500 and 1000, their query performance becomes slower. This is because our query processor expands the aggregated XML data into node tuples to report intermediate results, so many duplicate nodes are instantiated. For example, in **Figure 17**, a single a node in the hierarchical data is copied three times to generate intermediate node tuples. This inefficiency can be improved by holding intermediate results as a tree structure.

Our experiments demonstrate the scalability of our query optimization techniques to process queries of relatively large amount of results. If value conditions are involved, input data sizes of the amoeba join will be squeezed, so naive application of the amoeba join probably works well even for multiple input nodes. It still needs further study to estimate costs of amoeba join operations for various input data. Other than this cost estimation methodology, we can leverage the existing techniques of System R style query optimization on our XML algebra. In addition, the relational-style XML query provides independence of query statements from the underlying XML data structure. This property can be utilized to reorganize XML data structure for efficient query processing or minimizing database sizes. Although it might be possible to use relational databases as a storage scheme for relations in XML, it must have capabilities to query and store other XML nodes associated to relations.

7. RELATED WORK

The use of relational model to query complex structured data, including XML, has been studied in [16]. Our approach is unique in that it allows structural variations of XML data, and utilizes functional dependencies to capture data semantics of XML.

Finding Relations in XML. There have been several studies of the problem in finding relations in XML; Y. Li et al. [15] attempted

	Relation (Query Expression)	**FD**	**Query Schedule**
Q_1 Q_2	(site, person) (person, interest)	person → site interest → person	$AJ_{\text{site, person}}$(site, person) $AJ_{\text{person, interest}}$(person, interest)
Q_3 Q_{3F}	(site, person, interest) (site, person, interest)	 interest → person, person → site	$AJ_{\text{site, person, interest}}$(site, person, interest) $AJ_{\text{site, person, interest}}$($AJ_{\text{site, person}}$(site, $AJ_{\text{person, interest}}$(person, interest)))
Q_4 Q_{4F}	(regions, item, mail, date) (regions, item, mail, date)	mail → date mail → date, mail → item, item → regions	$AJ_{\text{regions, item, mail, date}}$(regions, item, $PC_{\text{mail, date}}$(mail, date)) $AJ_{\text{regions, item, mail, date}}$($AJ_{\text{regions, item}}$(regions, $AJ_{\text{item, mail}}$(item, $PC_{\text{mail, date}}$(mail, date)))
Q_5 Q_{5F}	(mail, from, date) (mail, from, date)	 mail → from date	$AJ_{\text{mail, date, from}}$(mail, date, from) $AJ_{\text{mail, from, date}}$($AJ_{\text{mail, from}}$($AJ_{\text{mail, date}}$(mail, date), from))
Q_6	(open_auction, current, (bidder, increase, time))		$AJ_{\text{open_auction, current, bidder, increase, time}}$(open_auction, current, bidder, increase, time)
Q_{6F}	(open_auction, current, (bidder, increase, time))	open_auction → current, bidder → open_auction, bidder → increase time	$AJ_{\text{open_auction, current, bidder, increase, time}}$($AJ_{\text{open_auction, current}}$(current, $AJ_{\text{open_auction, bidder}}$(open_auction, $AJ_{\text{bidder, increase}}$(increase, $AJ_{\text{bidder, time}}$(bidder, time)))))
Q_{6S}	(open_auction, current, (bidder, increase, time))	bidder → open_auction	$AJ_{\text{open_auction, bidder}}$($AJ_{\text{open_auction, current}}$(open_auction, current), $AJ_{\text{bidder, increase, time}}$(bidder, increase, time))
Q_7	(a, b, c)	c → b, b → a	$AJ_{\text{a, b, c}}$($AJ_{\text{a, b}}$(a, $AJ_{\text{b, c}}$(b, c)))

Figure 15: Query schedules for retrieving relations with several FDs in XMark (Q_1 to Q_{6S}) and synthetic data set (Q_7).

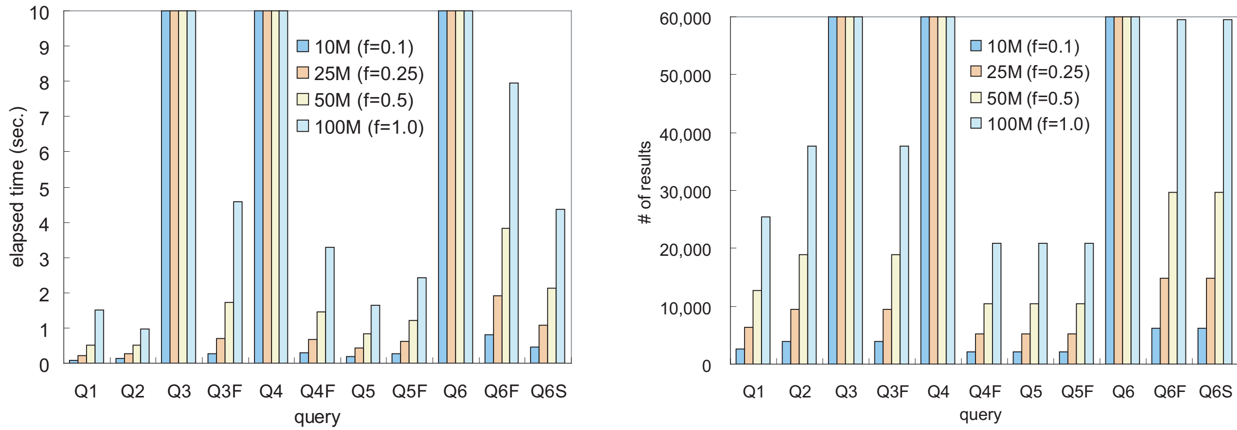


Figure 16: Query performance (Left) and result sizes (Right) of $Q_1 - Q_{6S}$. Performance and result sizes of Q_3, Q_4 and Q_6 could not be measured due to out of memory errors.

```
<table>
 <a value="1">
  <b value="1">
   <c value="1"/>
  </b>
 </a>
 <a value="1"/>
  <b value="2"/>
   <c value="2"/>
  </b>
 </a>
 <a value="1">
  <b value="2">
   <c value="3"/>
  </b>
 </a>
</table>
```

```
<table>
 <a value="1">
  <b value="1">
   <c value="1"/>
  </b>
  <b value="2">
   <c value="2"/>
   <c value="3"/>
  </b>
 </a>
</table>
```

a	b	c
1	1	1
1	2	2
1	2	3

```
<table>
 <b value="1">
  <c value="1">
   <a value="1"/>
  </c>
 </b>
 <b value="2">
  <a value="1">
   <c value="2"/>
   <c value="3"/>
  </a>
 </b>
</table>
```

Figure 17: Synthetic XML data of simple (left), hierarchical (center) and random (right) structures, generated from the same input table data.

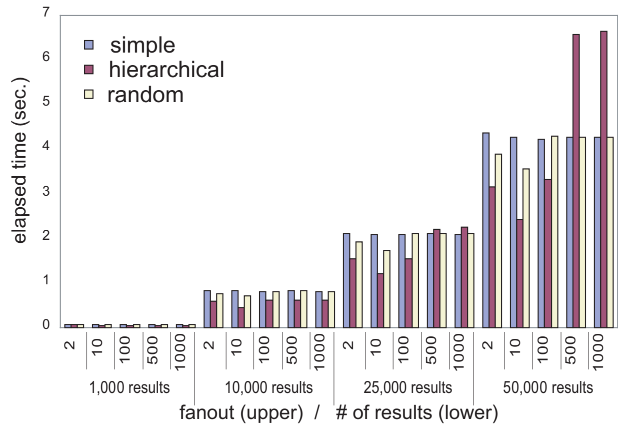


Figure 18: Query performance of Q_7 for variously structured XML data, which have the same number of relations (a, b, c).

to extract particular patterns, containing the *smallest least common ancestor (slca)* of a given set of XML nodes. The *slca*, which was coined in [27], is a least common ancestor (*lca*) that contains no other *lca* nodes among its descendants. This definition of *slca* is an attempt to exclude the XML root node from query results. This is because XML is a single-rooted tree, and thus irrelevant nodes that never belong to the same relation may be connected through the root node. However, the *slca* approach is highly dependent on the query input. For example, when two unrelated nodes are the inputs of an *slca* query, the root node will be wrongly reported as a query result. The *amoeba join* [19] successfully avoids such unintentional results, since it does not rely on any additional *lca* nodes. However, the cost of enumerating all tree structures is prohibitive without the knowledge of functional dependencies. Query methods that retrieve XML structures without using knowledge of the schema or FDs do return incorrect results. Several such cases were presented in [23].

Another approach to querying variously structured XML data is to search the data to the level of ancestor or descendant nodes [2, 10] or nearest neighbor nodes [26]. However, these methods cannot address all possible tree structures derived from relational data. In addition, they are optimized for keyword-search queries, and are thus not suited to rigid database queries.

Functional Dependencies for XML. FDs and keys have been well studied to find ways of reducing data redundancy and avoiding update anomalies [17]. In recent years, these concepts have been applied to XML in the form of XML keys [7] and XML FDs [3, 13, 24, 28]. These approaches are based on paths; given sets X and Y of paths, an FD for XML is defined as $X \rightarrow Y$. However, these path-based definitions of FDs cannot handle XML documents containing structural variations, which require multiple path expressions.

In summary, previous work on FDs for XML [3, 7, 13, 24, 28] inferred FDs from a path structure of an XML document. In contrast, our approach that assumes FDs are defined outside the XML data, and are specified using node names (e.g., tag or attribute names) on a relation, rather than on paths. Unlike path-based definitions, our definition of FD allows various XML data expressions, and therefore makes the design of XML databases much easier.

8. CONCLUSIONS

The presence of structural variations is a serious problem for the traditional XML query processors, because path-expression queries are dependent to the underlying XML tree structures. We overcome this problem by introducing the relational-style XML query, which uses the notion of a relation in XML that allows amoeba structures. In addition, to capture the data semantics implied in the XML structure, we incorporated the well-known notion of functional dependencies into XML, and devised efficient query processing techniques for retrieving relations satisfying FDs. With these capabilities, we can utilize heterogeneous XML structures to design and integrate several XML databases. The contributions described in this paper include:

- The notion of the relation in XML. With this capability, FDs and keys are smoothly incorporated into XML.
- A class of XML structures, called a tree relation, which can be used as an XML counterpart of relational tables.
- A departure from path-expression queries. XML structures of interest are automatically determined from a set of FDs.
- Capability of integrating variously structured XML data.
- Experimental results that confirm the scalability and tolerance of our query method in the presence of structural variations.

Repeatability Assessment Result

All the results (except Q_5 to Q_{6s}) in this paper were verified by the SIGMOD repeatability committee. Results of query Q_5 to Q_{6s} were added after the submission of the code in order to reflect a reviewer's comment. Code and data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/.

9. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] S. Amer-Yahia, L. V. Lakshmanan, and S. Pandit. FleXPath: Flexible structure and full-text querying for XML. In *SIGMOD*, 2004.

[3] M. Arenas and L. Libkin. A normal form for XML documents. In *ACM TODS*, 2004.

[4] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1).

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Floresch, J. Robie, and J. Simeon. XQuery 1.0: An XML query language - W3C working draft, November 2003. http://www.w3.org/TR/xquery.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition), October 2000. http://www.w3.org/TR/REC-xml.

[7] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. In *WWW*, 2001.

[8] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999. http://www.w3.org/TR/xpath.

[9] Extensible HyperText markup language (XHTML) 1.0 (second edition), January 2000. http://www.w3.org/TR/xhtml.

[10] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[11] H. V. Jagadish, L. V. S. Lakshman, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *DBPL*, 2001.

[12] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: One hierarchy isn't enough. In *SIGMOD*, 2004.

[13] M. L. Lee, T. W. Ling, and W. L. Low. Designing functional dependencies for XML. In *EDBT*, 2002.

[14] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, 2001.

[15] Y. Li, C. Yu, and H. V. Jagadish. Schema-free XQuery. In *VLDB*, 2004.

[16] T. H. Merrett. Aldat: A retrospective on a work in progress. *Inf. Syst.*, 32(4):505–544, 2007.

[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.

[18] RelaxNG. http://relaxng.org.

[19] T. L. Saito and S. Morishita. Amoeba join: Overcoming structural fluctuations of XML data. In *WebDB*, 2006.

[20] SAX: The simple API for XML. http://www.megginson.com/sax/.

[21] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolesch, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[22] Sleepycat Software. BerkeleyDB. http://www.sleepycat.com/.

[23] Z. Vagena, L. S. Colby, F. Özcan, A. Balmin, and Q. Li. On the effectiveness of flexible querying heuristics for XML data. In *XSym*, 2007.

[24] M. W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. In *ACM TODS*, 2004.

[25] XML schema. http://www.w3.org/XML/Schema.

[26] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *SIGMOD*, 2005.

[27] Y. Xu and Y. Papaconstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.

[28] C. Yu and H. V. Jagadish. Efficient discovery of XML data redundancies. In *VLDB*, 2006.