On the Information Content of an XML Database

Laks V. S. Lakshmanan∗
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
laks@cs.ubc.ca

Fereidoon Sadri†
Department of Mathematical Sciences
University of North Carolina at Greensboro
Greensboro, NC, USA
sadri@uncg.edu

Abstract

While XML arrived as a syntax for data exchange, its use in the database community has blossomed beyond its origins into a flexible data model for managing structured and semistructured data. However, the flexibility comes with the price of increased complexity. Querying XML data is much more complicated than querying a relational database, even when the underlying schema is available.

We contend that one of the root causes of this difficulty is that the information content modeled by an XML database has not been pinned down. In this paper, we make the following contributions. Using a notion of valuations from an XML database schema to the XML database, we capture the latter’s information content. We provide an alternative computational semantics for the information content and show it is equivalent to the valuation-based notion of information content. We show the information content can be used to faithfully reconstruct the original XML database. The information content can be represented as a relational database, which offers a high level information-preserving view of the original XML database. With this user interface, the user can write queries in SQL which can be translated into equivalent XQuery queries against the original XML database. We develop a translation algorithm and establish its correctness. Finally, to support flexible XML output construction, we propose a lightweight extension to SQL, called SQL-X. Our translation algorithm can correctly translate queries expressed in SQL-X. We also show by examples that a rich class of XQuery queries can be captured using the information content view as a front end and SQL(-X) as a query language. Examples show that complex and sophisticated XQuery queries against the XML database can be captured via simple and elegant SQL-X queries against the information content view.

1. Introduction

While XML arrived as a syntax for data exchange, its use in the database community has blossomed beyond its origins into a flexible data model for managing structured and semistructured data. Indeed, XML data model is much more flexible than the traditional relational model. The flexibility permits convenient and effective modeling of semistructured data arising in many applications like scientific data and data integration. However, the flexibility comes with the price of increased complexity. The schema of an XML database is much more complex. Querying XML data is much more complicated than querying a relational database, even when the underlying schema is available. The following example illustrates this.

Example 1.1 [Illustrating XML Querying] Consider an XML database conforming to the following DTD, where sid, state, city, year, month, and sales are PCDATA.

```
<!ELEMENT salesinfo (store*)>
<!ELEMENT store (sid, state, city, sale*)>
<!ELEMENT sale (year, month, sales)>
```

The query “list total annual sales for stores grouped by state, city, store, year for the years 1990-2003” can be expressed in XQuery as follows.

```
for $St in distinct-values(doc(...)//state),
  $Ct in distinct-values(doc(...)//store[state=$St]/city),
  $Y in distinct-values(doc(...)//store[state=$St and city=$Ct]/sale/year,
      where $Y >= 1990 and $Y <= 2003
  let $S := doc(...)//store[state=$St and city=$Ct]/sale/year=$Y]/sales
return
  {$St}, {$Ct}, {$Y}, {sum($S)}
```

For writing this query, it is critical that we know how the salesinfo is represented, specifically how elements are nested and whether any references are used for linking information. In particular, if either the nesting structure is altered (e.g., stores are nested inside cities which are in turn nested inside states) or if some of the association is captured via references (e.g., location information (state, city) and store sales information (year, month, sales) are represented
as separate elements with an IDREF links from stores to the respective elements), then formulation of the above query would be substantially different.

One of the motivations behind this work is to simplify the task of query formulation against XML data. As we shall show, the formal underpinning behind this desired simplification is the task of pinning down exactly the information content of an XML database. As a preview, let us pretend that there was a relation $V$ capturing all the information in the XML database above, with attributes corresponding to the leaf elements of the XML database. E.g., $V$ has attributes sid, state, city, etc... Then the above query could be formulated in SQL simply as

```sql
select state, city, year, SUM(sales)
from V
where year <=2003 and year >=1990
group by state, city, year
```

Clearly, the SQL query formulation is considerably simpler. Furthermore, if our relation $V$ could capture the information of the XML database no matter how the information was structured (e.g., with different nesting structure) but with same element names, then the SQL query would remain the same, although the corresponding XQuery statement would be considerably different.

There are three issues here. (1) Where do we look for this relation (or more generally set of relations) $V$ which exactly captures the information content of the XML database? (2) Given that it is unrealistic to materialize $V$ in general, we need a mechanism for translating SQL queries against $V$ to corresponding XQuery statements against $V$. (3) Finally, XQuery offers considerable flexibility in output construction whereas SQL produces a flat output. We need a mechanism for flexible output construction.

In this paper, we make the following contributions.

- Using a notion of valuations from an XML database schema to the XML database, we capture the latter’s information content. (Section 2.2)

- We provide an alternative computational semantics for the information content and show it is equivalent to the valuation-based notion of information content (Section 3).

- We show the information content can be used to faithfully reconstruct the original XML database. (Section 3.2)

- Thus, the information content can be represented as a relational database, which offers a high level information-preserving view of the original XML database. With this user interface, the user can write queries in SQL which can be translated into equivalent XQuery queries against the original XML database. We develop a translation algorithm and establish its correctness (Section 4).

- Finally, to support flexible XML output construction, we propose a lightweight extension to SQL, called SQL-X. Our translation algorithm can correctly translate queries expressed in SQL-X. We also show by examples that a rich class of XQuery queries can be captured using the information content view as a front end and SQL(-X) as a query language. Examples show that complex and sophisticated XQuery queries against the XML database can be captured via simple and elegant SQL-X queries against the information content view (Section 5).

The relevant background and definitions appear in Section 2.1. Section 6 discusses related work. In Section 7, we summarize the paper, discuss the vision for this project and future work.

## 2. Capturing XML information content

In this section we will introduce the notion of valuations, mappings from XML database schema to XML database contents, and argue they represent the information contained in the database. Then we represent an alternative syntactic characterization of XML information content, called Information Content Tableaux (ICT), in Section 3, and show the equivalence of ICT and set of valuations. We start with some definitions.

### 2.1. Definitions

Our starting point is an XML document, plus its schema. The schema of an XML document can be specified in a suitable language (such as DTD or XML Schema), and may be readily available or can be derived (discovered) from the document itself [9, 10]. Rather than using a schema definition language, in this paper, we will use a graphical representation that captures the structural aspects of the schema.

**Definition 2.1 (Schema Graph)** A schema graph is an edge- and node-labeled directed graph $G = (V, E)$ where $V$ is the set of vertices: For each element or attribute there is a vertex in $G$ labeled by the element tag or the attribute, and $E$ is the set of edges, where $(v_i, v_j) \in E$ if $v_j$ represents a subelement or an attribute of the element $v_i$. Edges are labeled by one of the quantifiers ‘1, ?, +, *’ with their obvious meaning.

A schema graph may be a tree, a directed acyclic graph, or may even contain cycles (because of recursion). It may
Furthermore, we have disjunction, but we do not consider disjunction in this paper. We will make use of the notion of schema tree throughout the paper. For this purpose, we make use of the following DTD transformation. Schema trees play an essential role in pinning down the information content.

**Algorithm 2.1 (DTD Transformation)**

Input: DTD $D$.
Output: A DTD $D''$, derived from $D$, whose graph representation is a tree.

1. Mark all tags in $D$ that participate in a cycle.
2. Arrange all unmarked tags in a partial order: $A$ precedes $B$ iff $A$ is reachable from $B$. (This is well defined since neither of $A, B$ participates in a cycle.)
3. Visit unmarked tags in their precedence order.
4. For each unmarked tag $B$ s.t. $B$ appears in the RHS of $k > 1$ productions: $A_1 \rightarrow B \cdots$; $A_k \rightarrow B \cdots$ & all unmarked tags preceding $B$ have been processed { 
   Rename the RHS appearances with new tags: $A_1 \rightarrow B_i \cdots$, $i = 1, \ldots, k$; 
   Replace the production (if any) $B \rightarrow \alpha$ with the $k$ productions $B_i \rightarrow \alpha_i$, $i = 1, \ldots, k$, where $\alpha_i$ is $\alpha$ with tags $C$ in it replaced by $C_i$. 
   Do this production replacement recursively. 
   Flag $B$ as processed. }
5. Consider the productions of the resulting dtd $D'$ in the following order.
   the root-production is considered first. Leave the root-production intact and flag all tags in this production as "seen".
   Before considering a production $A \rightarrow \alpha$, make sure $A$ is already marked as "seen".
   When we consider this production, if any tag $B$ in $\alpha$ is already seen, rename it to a new label $B'$. 
   Note that no new productions with LHS $B'$ are added. 
   Let $\alpha'$ be the result of renaming all such tags in $\alpha$ to their new names. Flag all tags in $\alpha'$ as "seen".
   Repeat this until all productions in $D'$ have been considered.
6. The resulting DTD is $D''$.

For a given DTD $D$, it can be shown that $D''$ is a DTD whose graph representation is a tree (see example below). We call the graph representation of $D''$ the schema tree of $D$. It can also be shown that when $D$ is DAG, $D''$ is equivalent to $D$, i.e., both represent the same language, although $D''$ uses more intermediate nodes.

**Example 2.1 (Schema tree)** Consider the following schema:

```
<!ELEMENT dept (faculty*)>
<!ELEMENT faculty (name, ra*, ta*, paper*)>
<!ELEMENT paper (section*)>
<!ELEMENT section (title, section*)>
<!ELEMENT ra (name)>
<!ELEMENT ta (name)>
```

Schema graph $G$ and schema tree $T_G$ are shown in Figure 1. We have used descriptive labels $fname$, $rname$, $tname$, and $subsection$ in $T_G$ (instead of indexed labels).

![Figure 1. Schema graph, and corresponding schema tree](image)

An XML document (instance) can be regarded as a tree, even when its schema is a DAG or is cyclic. Our definition of document tree is similar to the document model used in the XPath specification [18]. The major difference between this model and DOM (Document Object Model) [8] is that we treat attributes as children of the elements bearing them.

**Definition 2.2 (Document Tree)** The document tree $D(V, E)$ is a rooted ordered tree with set of nodes $V$ and set of edges $E$. Each node $v \in V$ represents an element or an attribute, and is labeled by the element tag or the attribute name. We also assume each node has a unique identifier (which could simply be the pre-order number of the node). An edge $(v_i, v_j) \in E$ if $v_j$ is a subelement or an attribute of $v_i$.

**Example 2.2** The document tree corresponding to a document on the schema of Example 2.1 is shown in Figure 2. We have used a letter to specify node types when needed. For example, $3r$ is an $ra$ node, while $5p$ is a $paper$ node.

![Figure 2. Document tree](image)

Note that a document tree only represents elements and attributes, but not their content or values. We assume a value function that takes a node identifier and returns the content of the element or value of the attribute.

**Definition 2.3 (The value function)** The value function is a function from the set of document tree nodes $V$ to the set
of values defined as follows: If the node is a leaf, it is either an attribute (having a value) or an element (having a content but no subelements), and the value function returns attribute value or element content. For an internal node \( n \), we extend the value function to return the ordered list of values of \( n \)'s children, recursively. Mixed content, i.e., elements having subelements and (possibly interspersed) contents, can be easily incorporated in the fashion of TIMBER [13] by introducing children of type “content” to account for the content(s) of internal nodes.

2.2. Valuations

In this section, we define valuations. Intuitively, our goal in capturing the information content of an XML database (document) is to capture what is the set of atomic propositions that are supposed to be true in the database, from the viewpoint of associations between values. We do this by means of valuations, which are mappings from a schema tree to a database that preserve associations.

Definition 2.4 (Valuation) Given an XML document \( D \), its document tree \( T_D \), its schema graph \( G \), and its schema tree \( T_G \), a valuation \( \nu \) is a (possibly partial) mapping from the nodes of \( T_G \) to the nodes of \( T_D \) defined as follows: (1) Whenever \( \nu(n) \) is defined on a schema tree node \( n \), \( \nu(n) \) and \( n \) have the same tag or represent the same attribute. (2) The mapping \( \nu \) preserves element/subelement and element/attribute relationships, i.e., if \( n \) is the parent of \( m \) in \( T_G \) and if \( \nu(n) \) and \( \nu(m) \) are defined, then \( \nu(n) \) is the parent of \( \nu(m) \) in \( T_D \), and (3) The mapping \( \nu \) satisfies the following: If \( n \) is the parent of \( m \) in \( T_G \) and \( \nu(n) \) is defined, then \( \nu(m) \) can be undefined only if \( \nu(n) \) does not have any children corresponding to \( m \). Similarly, if \( \nu(m) \) is defined, then \( \nu(n) \) can be undefined only if the parent of \( \nu(m) \) does not correspond to \( n \). We also require a valuation to be defined on at least one node of the schema tree.

Condition (3) is needed to ensure empty valuations are illegal, as well as to ensure a valuation is undefined on a schema tree node, only when it needs to be. It also helps handle cyclic DTDs correctly.

Example 2.3 (Valuation) Consider the schema tree of Figure 1 and document tree of Figure 2. There are four valuations for this example. We will only show two of them. The first valuation maps nodes dept, faculty, fname, ra, rname, paper, section, title, subsection to document tree nodes (0, 1, 2, 3, 4, 5, 6, 7, 8), respectively, and mappings for nodes ta, tname are undefined. This basically captures information on faculty (represented by node) 1, which includes the top level of the recursion regarding paper section (represented by node) 6 (hence, 6 has title 7 and subsection 8). A second mapping only maps section, title to document tree nodes (8, 9), respectively, and is undefined for other schema tree nodes. This second valuation captures the second level of recursion, recording the title of section 8. Note that if section 8 had any subsections, this valuation would be defined for subsection (say, node 8a), and there would be other valuations capturing information on section 8a (title, possible subsections) and so forth.

Definition 2.5 (Subsumption) Given two valuations \( \nu_1 \) and \( \nu_2 \), we say \( \nu_1 \) subsumes \( \nu_2 \), \( \nu_1 \succeq \nu_2 \), if for all nodes \( n \) where \( \nu_2(n) \) is defined, \( \nu_1(n) \) is also defined and \( \nu_1(n) = \nu_2(n) \). It is easy to verify that \( \succeq \) is a partial order.

We show that valuations are maximal with respect to subsumption for XML with acyclic schemas. This, we argue, shows that each valuation captures as much information as possible on related elements and their attributes.

Theorem 2.1 Given XML document \( D \) with an acyclic schema, let \( \mathcal{V} \) be the set of all valuations from schema tree to document tree. Then every valuation \( \nu \in \mathcal{V} \) is maximal with respect to subsumption ordering.

Proof: First, it is easy to show that for documents with acyclic schema graph all valuations must be defined for the root of the schema tree. This is a consequence of condition 3 of valuations. To show valuations are maximal, we should show that for a pair of (distinct) valuations \( \nu_1, \nu_2 \in \mathcal{V} \), we can not have \( \nu_1(n) = \nu_2(n) \) for all \( n \) where \( \nu_2(n) \) is defined. Assume such \( \nu_1, \nu_2 \) exist. Then there should be a node \( m \) in the schema tree where \( \nu_1(m) \) is defined while \( \nu_2(m) \) is not. By condition 3, for all ancestors \( m_i \) of \( m \), including the root of the document tree, \( \nu_1(m_i) \) should be defined. It follows from condition 3 that \( \nu_2 \) should also be defined for all these nodes, including \( m \). This is a contradiction.

On the other hand, when schema is cyclic, we need to prevent valuations to mix unrelated data. This can be achieved by requiring one additional condition on valuations that we will discuss further below.

Case of cyclic schema
When schema graph is cyclic, there is a possibility that unrelated data from different cycles get mixed up in a valuation. For example, consider a DTD involving the following elements (among others)

```
<!ELEMENT part (... subpart*)>
<!ELEMENT subpart (part)>
<!ELEMENT person (... child*)>
<!ELEMENT child (person)>
```

A single valuation may map part, subpart, part as well as person, child, person to the instance, while there is no direct relationship between the part and the person. This is undesirable. Note that the same mapping of part, subpart, part (with person, child, person mapping undefined) is also a valuation under Definition 2.4, as is the same mapping of part, child, person (with part, subpart, part mapping undefined). In fact, the latter two mappings are the desired valuations. We define the set of valid valuations as follows to remedy mixing unrelated information in a valuation:

**Definition 2.6 (Valid valuations)** Let \( \mathcal{V} \) be the set of valuations under Definition 2.4. The set of valid valuations are the set of minimal elements of \( \mathcal{V} \) with respect to subsumption.

Note that Condition 3 in Definition 2.4 is needed to “pack as much related information as possible” into a valuation. While Definition 2.6 is needed to “avoid packing unrelated information” into a valuation, in the case of cyclic schemas. Our thesis then is that the information content of an XML database is the set of valuations.

**3. The Information Content Tableux**

We present Information Content Tableaux, ICT, a relational (tabular) representation that captures the information contained in an XML document. Although ICT can be materialized and used directly for processing queries, our main goal in this paper is to use ICT as a view to be used by users to query the XML document. Formulating queries on this simple view is, in general, significantly simpler than writing queries directly on the XML database. A user query, written on ICT, is translated by the system to a query on the underlying XML document, optimized, and evaluated.

**3.1. Capturing hierarchical relationships**

First, let us consider XML databases where relationships between objects are represented only by the hierarchical (element/sub-element or element/attribute) structure. This corresponds to XML documents that do not have any attributes of type IDREF/IDREFS nor utilize the XML Schema keyref mechanism. We use a relational representation, called the Hierarchical Information Content Tableau, HICT, to capture the information content of XML documents utilizing only hierarchical structure to represent relationships.

**Schema of HICT**

Consider an XML database \( D \) with the schema graph \( G \). Let \( T_G \) be the schema tree of \( D \). The hierarchical information content tableau, HICT, has one column for each node in \( T_G \), labeled by the label of the \( T_G \) node. That is, (relational) schema of HICT is the set of node labels (i.e., attributes and tags) in \( T_G \).

**Contents of HICT**

Let \( D, G, \) and \( T_G \) be as above, and let \( T_D(V, E) \) be the document tree of \( D \), with set of nodes \( V \) and set of edges \( E \). For each edge \( (v_i, v_j) \) in the schema tree \( T_G \) we define a binary relation

\[
r_{ij} = \{(p, q) \mid (p, q) \in E \text{ is an instance of } (v_i, v_j)\}
\]

where \( p \) and \( q \) are identifiers of nodes in the document tree \( T \). Then HICT is the full outer join of the set of binary relations corresponding to \( T_G \) edges:

\[
HICT = \bigotimes_{(v_i, v_j) \in T_G} (r_{ij})
\]

**Example 3.1 (Hierarchical ICT)** Consider the XML document represented by document tree of Figure 2, which corresponds to the schema graph and schema tree shown in Figure 1. The following demonstrates the construction of HICT. We have abbreviated column names in HICT, keeping only the first two characters of the column name.

<table>
<thead>
<tr>
<th>HICT:</th>
</tr>
</thead>
<tbody>
<tr>
<td>de</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Tables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>dept</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>ta</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>section</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

HICT is...
We would like to point out that, as we will prove later, tuples of the HICT of an XML document correspond to the valuations of that document. Readers can verify that the two valuations discussed in Example 2.3 correspond to the first two tuples of the HICT in this example.

The following example shows how HICT can be used for querying the XML document. Translation and optimization of queries are discussed in later sections.

Example 3.2 (Querying the HICT) The query “List faculty for whom “John Davis” works as an ra” is written as follows:

```sql
SELECT value(distinct(fname))
FROM HICT
WHERE value(rname)='John Davis'
```

Note the use of the `value` function in the query to map node identifiers to their values. The `distinct` function is also needed to avoid multiple listings. We will simplify the user interface later (Section 4) by adapting certain conventions that allow us to write the above query simply as

```sql
SELECT fname
FROM HICT
WHERE rname='John Davis'
```

3.1.1 A note on grouped subelements

Consider the following schema fragments (DTDs):

```xml
<!ELEMENT dept (dname, student*)>
<!ELEMENT student (id, sname)>
```

and

```xml
<!ELEMENT dept (dname, (id, sname)*)>
```

Both schemas can be used to represent a department and its students. In the first schema, the element `student` is used as a subelement of `dept` to represent each student. In the second schema the student information, namely, `(id,sname)` pairs, are direct subelements of `dept`. We contend that the first schema is a better schema to represent this information\(^2\). Notwithstanding, the information content tableau approach should be able to represent XML information conforming to either schema. To achieve this objective, we introduce virtual grouping elements to transform the schema (DTD) to one that does not have any subelement groups (a subelement group has the form `(element-list)*` or `(element-list)+` where `element-list` contains more than one element.)

For example

```xml
<?ELEMENT a (b, (c,d)*)>
```

is converted to

```xml
<?ELEMENT a (b,v*)  
<?ELEMENT v (c,d)>
```

where `v` is a (new) element (virtual grouping element). This process may need to be applied recursively if the schema contains nested subelement groups. Then, an intermediate ICT is obtained using the new schema. Finally, the columns representing the virtual elements are projected out to obtain the final ICT.

3.2. Properties of the hierarchical information content tableau (HICT)

Theorem 3.1 (Reconstructing original document) If node identifiers of the document tree are in document order, then the original document can be reconstructed from HICT, schema information, and value function.

Proof: We present an algorithm that generates the original document one element at a time. Proof that the generated document is identical to the original document is straightforward and is omitted.

Algorithm 3.1 (Reconstructing original document)

1. Using the schema tree, determine the schemas of the binary edge relations.
2. Project HICT over these schemas, eliminate duplicates and tuples with null values. These will be the original binary relations.
3. Generate the root element of the document. Determine its tag (from schema) and its corresponding node identifier in the document tree (from any binary relation with (root,child) schema).
4. Recursively, for each element in the document that has not been processed yet, generate its attributes and subelements, as follows:

Generating subelements and attributes of a given element

For each generated element, we know its tag, and its corresponding node identifier in the document tree. Given an element `e` with corresponding node id `i`, if node `i` is a leaf then use the value function to obtain the content of `e` and incorporate it into the element. Otherwise, determine the tags of (potential) children of `e` from the schema. Then use the corresponding binary relations to determine and generate all subelements (if any) for each child tag. The ordering of

\(^2\)Research on XML design is in its infancy, but there is significant interest in developing a comprehensive design theory for XML.
subelements is determined by the order of their node identifiers. If the child is an attribute, obtain the value of the attribute using the value function and generate the appropriate attribute=value expression inside the opening tag of e.

3.3. Properties of Valuations

We have provided two approaches for capturing the information content of XML data: A semantic approach based on the notion of valuations, and a syntactic approach based on combining (via outer-joins) parent/child relationships. In this section we will show the equivalence of these two approaches.

Theorem 3.2 (HICT equals set of valuation) Given a document with an acyclic schema graph, the tuples of HICT correspond exactly to the set of valuations.

Proof: We would like to show that each tuple of HICT corresponds to a valuation and vice versa. Let \( t \in HICT \) be a tuple in HICT. Recall that attributes of HICT correspond to nodes of schema graph. Define a mapping \( \nu \) where \( \nu(n) = t(n) \) if \( t(n) \) is non-null, otherwise \( \nu(n) \) is undefined. We claim that \( \nu \) is a valuation. Recall that \( t \) is a tuple in the outer join of the binary relations that correspond to the edges of the schema tree. The mapping \( \nu \) obviously satisfies conditions (1) and (2) of valuations (Definition 2.4) since the values of the binary relations are of the right type and preserve hierarchy. Condition (3) is also satisfied since if \( m \) is a node in schema tree, (a) the binary relation \( r(p,m) \), where \( p \) is the parent of \( m \) in schema tree, will have an entry \( (c,t(m)) \), for some value \( c \) corresponding to node \( p \). Then \( t(p) = c \). Hence \( \nu(p) = c \). (b) Similarly, if \( q \) is a child of \( m \), then \( r(m,q) \) either has an entry \( t(m), d \), or it does not. In the first case, \( t(q) = d \) and \( \nu(q) = d \). In the second case, \( t(q) \) is NULL and \( \nu(q) \) is undefined. The second case corresponds to the situation that \( \nu(m) \) does not have a child of type \( q \), since \( t(q) \) is NULL.

For the converse, let \( \nu \) be a valuation, and define a tuple \( t \) by \( t(n) = \nu(n) \) if \( \nu(n) \) is defined, otherwise \( t(n) \) is NULL. We claim that \( t \in HICT \). \( t \) can be generated by starting at the root and climbing down the tree (in preorder, or level-order) performing outer joins with the binary relations corresponding to the edges of the schema tree. This follows the facts that \( \nu(root) \) is defined so \( t(root) \) is non null. Further, if \( (n,m) \) is an edge in the schema tree and \( \nu(n) \) and \( \nu(m) \) are defined, then the binary relation \( r(n,m) \) contains \( (\nu(n),\nu(m)) \). Further, if \( \nu(m) \) is undefined for a node \( m \), then \( \nu(n) \) is also undefined for all nodes in the subtree rooted at \( n \). On the other hand, \( t(m) \) is NULL in this case, and, as the effect of outer join, \( t(m') \) will also be NULL for all nodes \( m' \) in the subtree rooted at \( m \). It follows that \( t \) is generated in the outer join of the binary relations corresponding to the edges of the schema tree, hence, \( t \in HICT \).

Case of cyclic schema

We presented the notion of valid valuations for XML data with possibly cyclic schemas (Section 2.2). Theorem 3.2 can be generalized to show that HICT corresponds exactly to the set of valid valuations (for acyclic as well as cyclic schemas).

Theorem 3.3 (General case) Given an XML document, the tuples of HICT correspond exactly to the set of valid valuations.

Proof sketch: If the schema is acyclic, valuations are all maximal and hence valuations and valid valuations coincide. Proof immediately follows from Theorem 3.2.

If the schema is cyclic, let \( n_1, \ldots, n_k, n_1 \) be a cycle in schema graph \( G \). The cycle is broken by removing the edge \((n_k,n_1)\) from \( G \), and adding a new node \( n'_1 \), and a new edge \((n_k,n'_1)\) to obtain the schema tree \( T_G \). Let \( n \) be the parent of \( n_1 \) in the schema tree. We notice that mappings \( \nu \) of \( n_1, \ldots, n_k, n'_1 \) to the document tree can have two forms: (1) Mappings where \( \nu(n) \) is also defined. These mappings correspond to the first (topmost) elements in the recursive chain, and (2) Mappings where \( \nu(n) \) is undefined. These mappings correspond to all subsequent elements in the recursive chain. Valid valuations with mappings of type (1) can be shown to be in HICT using Theorem 3.2. For valid valuations with type (2) mappings, we notice that only nodes in the subtree of schema tree rooted at \( n_1 \) can have defined mappings, and mappings for all other nodes has to be undefined. This follows from Condition (3) of Definition 2.4 plus Definition 2.6. It is easy to verify that for nodes of the document tree corresponding to elements other than the topmost elements in the recursive chain, the outer-join generates tuples that exactly match these type (2) valid valuations.

3.4. Capturing Order

Order is an important feature of XML data model for certain applications. XML query languages have built-in functionalities to query order-related features of XML data, and represent the result in document order. For example, a query may ask for the third figure in fourth section of second chapter. On the other hand, the relational model does not have any native order-related capabilities. Approaches to capture the order of XML data in the relational world have been proposed recently (for example, [7, 17]). These proposals use additional attributes in the relations to capture order information the original XML document.

As shown in Theorem 3.1 the hierarchical information content tableau, HICT, captures the information content of an XML document faithfully, in the sense that the original document can be reconstructed. However, it is difficult (and sometimes impossible without imbedding in a general
purpose programming language) to use SQL to formulate order-related queries. We use a simple approach for capturing order information in ICT that can be used easily by SQL. We should emphasize that ICT is not intended to be materialized and used directly for query processing. Rather, it is used as a view upon which user queries are formulated. The system than translates this query back to XML, optimizes the translated query, and executes it. As such, we are not seeking the most efficient order representation approach. Rather, we would like the simplest approach from query formulation view.

To capture order information, we modify HICT in the following way. Recall that HICT was defined to be the outer join of the (binary) relations corresponding to the edges of schema tree (Section 3.1). We add an extra column to each of these relations. Hence, for each edge \((v_i, v_j)\) in the schema tree \(T_G\) we define a ternary relation \(r(v_i, v_j, v_{ij}\text{pos})\) that, in addition to \((p, q)\) node pairs in the document tree corresponding to \((v_i, v_j)\), records the position of the \(q\) node among its siblings. Then, HICT is the full outer join of these relations.

This (position) information, together with the schema tree \(T_G\), can determine the document order of the XML data. Note that the position information alone, called Same-Sibling Order Encoding in [17] is not sufficient to recreate document order.

Example 3.3 Consider the XML document of Example 3.1. HICT (with position attributes included) will have, among others, faculty-pos, ta-pos, and ra-pos columns. The query list the second ra of “Mary Thomas” is written (in the simplified syntax) as

```sql
select rname
from HICT
where fname='Mary Thomas' and ra-pos=2
```

As another example, list the second ra of the fifth faculty is written as

```sql
select rname
from HICT
where faculty-pos=5 and ra-pos=2
```

Translation of queries to XQuery is discussed in Section 4.1.4. See Example 4.4 for the XQuery translation of the SQL query above.

3.5. Capturing relationships represented by (ID and key) references

Relationships may be represented by IDREF/IDREFS and keyref mechanisms in XML as the following example demonstrates:

Example 3.4 (Schema with IDREFS attribute) The information in Example 3.1 can use the following schema:

```xml
<!ELEMENT dept (faculty*,student*)>
<!ELEMENT faculty (name)>
<!ELEMENT student (name)>
<!ATTLIST faculty ra IDREFS, ta IDREFS>
<!ATTLIST student sid ID>
```

In this schema, the faculty-ra and faculty-ta relationships are represented by IDREFS attributes 'ra' and 'ta' that refer to student's ID attribute 'sid'. This representation is closer to the database schema design principles, and avoids redundant student data that could exist in the presentation of Example 3.1.

The linked information is already captured by HICT through the value function that can be used to obtain attribute values. However, SQL is inadequate to handle set-valued attributes such as IDREFS. For this reason, we use separate relational representations, called the Linked Information Content Tableau, LICT, to capture the portion of the information content of the XML data that is represented through references. Hence, for the purpose of querying, the information content of an XML document is captured by a single hierarchical ICT (HICT) that captures the hierarchically represented data, plus zero or more linked ICT (LICT) that capture the information represented through references.

Linked Information Content Tableaus, LICT

There is an LICT for each pair of (IDREF(S), ID) attributes. We consider the link to be from the IDREF attribute to the parent of the ID attribute.

Consider an XML database \(D\) with schema tree \(T_G\) and document tree \(T_D\). Let nodes \(n_1\) and \(n_2\) in \(T_G\) correspond to an IDREF(S) to parent-of-ID link. Minimally, the LICT corresponding to \((n_1, n_2)\) link, LICT\((n_1, n_2)\), should consist of the binary link relation \(r(n_1, n_2)\):

\[ r(n_1, n_2) = \{(p, q)\mid (p, q) \text{ is an instance of } (n_1, n_2) \text{ link} \} \]

That is, \(p\) and \(q\) are identifiers of nodes in document tree corresponding to \(n_1\) and \(n_2\), and \(p\) refers to \(q\).

The link relation is adequate (in conjunction with HICT) to formulate queries on linked relationships. But, almost all such queries will involve join of link relation with HICT. To facilitate user query specification, we define a richer LICT, which can be used to formulate a wide spectrum of queries with no need for joining with HICT. We first need some definitions.

3 In XML, IDREF(S) attributes are untyped. This means an IDREF(S) attribute can refer to any ID attribute value. In practice, usually, IDREF(S) are used to represent relationships between two entity sets. Hence, a specific IDREF(S) attribute often takes values from a single ID attribute. Hence, For each IDREF(S) attribute, there is usually a single nonempty LICT, although it is possible to have more than one in some cases.
Definition 3.1 (Subtrees corresponding to a link) Let \( n_1 \) and \( n_2 \) be the same as above. Let node \( n \) be the least common ancestor of \( n_1 \) and \( n_2 \) in \( T_G \). Let \( n_1' \) and \( n_2' \) be the children of \( n \) that are on the paths from \( n \) to \( n_1 \) and \( n_2 \), respectively. Consider (1) subtree of \( T \) rooted at \( n_1' \) (let's call this \( T_1 \)), and (2) subtree of \( T \) rooted at \( n_2' \) (let's call this \( T_2 \)). We call \( T_1 \) and \( T_2 \) subtrees (of the schema tree) corresponding to \((n_1, n_2)\) link. ■

Note that \( T_1 \) and \( T_2 \) are disjoint, \( n_1 \in T_1 \), and \( n_2 \in T_2 \). For example, consider the DTD of Example 3.4, whose schema tree appears in Example 4.2. There are two links: \((\text{ra, student})\) and \((\text{ta, student})\). \( T_1 \) and \( T_2 \) corresponding to the first link contain the nodes \((\text{faculty, fname, ra, ta})\), and \((\text{student, surname, sid})\), respectively.

Now we can describe the construction of the LICT.

Definition 3.2 (LICT) The schema of \( \text{LICT}(n_1, n_2) \) corresponds to the nodes of \( T_1 \) and \( T_2 \). Let \( r_1, \ldots, r_k \) be the ternary edge relations, as discussed in Section 3.4, corresponding to edges of \( T_1 \) and \( T_2 \). Then \( \text{LICT}(n_1, n_2) \) is the outer join of the link relation \( r(n_1, n_2) \) with the edge relations \( r_1, \ldots, r_k \). ■

Example 3.5 (Query using LICT) Assume the relationship about faculty ra’s is represented by IDREFS link. The query “List faculty for whom “John Davis” works as an ra” is written as follows:

```
select fname
from LICT(ra, student)
where sname='John Davis'
```

Note that the query is almost identical to the same query on hierarchically represented relationship of Example 3.2. ■

4. Querying XML information through ICT

Our goal if to facilitate expressing queries on XML documents. To formulate a query directly on an XML document (for example, in XQuery), users require detailed knowledge of the structure of the document. Using information content tableaux, users only need a knowledge of element and attribute names. Further, relational query languages are significantly simpler than XML languages such as XQuery.

When relationships in an XML document are represented hierarchically as well as through (IDREF, keyref) links, that information is also needed to formulate queries on ICT correctly.

We also assume that the distinct and value functions are applied by default to user queries. More specifically, the distinct function is applied to the select clause first, and value function is applied to all attributes that represent leaves of schema tree. Of course, if we wanted to materialize ICT tableaus and evaluate the query directly, then we would need a preprocessor to add distinct and value functions to the user query, and then evaluate it against ICT. But for translating the user query to a query on the XML document, the simplified syntax is at once adequate and more friendly.

We should also mention here that an SQL query on ICT defines a flat output. A query language capable of defining output structure is needed to formulate queries with an XML output. For data exchange purposes we could by default wrap this flat output minimally to conform to XML syntax. Further, we will present a lightweight extension to SQL to support XML structuring in Section 5.

Examples

Consider an XML database consisting of books, publishers, and authors, as well as other relevant information such as book price, title, pubdate (publication date), etc. Consider the query “list titles of books published by Prentice Hall in 2004”. As long as we know the relationship between books, their titles, publishers, and pubdate are represented hierarchically (but in whatever way), we can formulate this query against ICT in SQL as:

```
select title
from HICT
where publisher = 'Prentice Hall' and
      publish-date = '2004'
```

In particular, regardless of whether publishers are nested inside books which are inside authors or publishers and authors are both nested inside books, or some other hierarchical organization is used, our query formulation above remains unaffected. Besides, the amount of schema knowledge required for query formulation is minimal compared with formulating a query against the XML database directly. As we will see in Section 4.1, our translation algorithm translates the query correctly to the appropriate XML database schema.

Suppose now that the association between books and authors is represented via IDREFS, while other associations are hierarchical. Then, the query “list author names and the number of books written by them” can be expressed against ICT as:

```
select a-name, count(book)
from LICT(authors, author)
group by a-name
```

The only additional information needed to formulate this query is that the element (or attribute) a-name, and book
4.1. Translating to XQuery

In this section we will present algorithms to translate and optimize a user query on ICT (HICT and/or LICTs) into an XQuery over the original XML document. We will begin with an algorithm for the translation of user queries that have a single HICT in the from clause. Then we will extend the algorithm to queries with LICT, as well as multiple HICT/LICTs in the from clause. We will also study queries involving aggregation and group-bys.

4.1.1 Translating a query on HICT

Let \( Q \) be a user query of the following form:

\[
\begin{align*}
\text{select} & \quad \text{target-list} \\
\text{from} & \quad \text{HICT} \\
\text{where} & \quad \text{conditions}
\end{align*}
\]

The input to the translation algorithm consists of the query \( Q \), plus schema information (schema tree). Let \( \mathcal{A} \) be the set of attributes that appear in \( Q \). That is, \( \mathcal{A} \) is the set of attributes in the select-clause \( \text{target-list} \) and in where-clause \( \text{conditions} \). Each attribute \( A \in \mathcal{A} \) corresponds to a node in schema tree \( T \). We denote this node by \( n_A \), and the set of nodes corresponding to the set of attributes in the query by \( N_A = \{ n_A \mid A \in \mathcal{A} \} \). The translated XQuery has the form:

\[
\text{(variable declarations)} \\
\text{where (translated conditions)} \\
\text{return (translated target-list)}
\]

The algorithm to generate variable declarations is the main part of the translation algorithm and is given further below. For each attribute \( A \) in the query, translated XQuery will have a variable \( \$A \). Variable declarations may involve additional auxiliary variables (see algorithm below). Translated conditions are obtained from the (SQL) where-clause conditions by replacing each attribute \( A \) by its corresponding XQuery variable \( \$A \). Translated target list is also obtained simply by replacing each attribute in the SQL target list with its corresponding XQuery variable.

**Generating XQuery variable declarations**

Intuitively, variables are declared using auxiliary variables that represent their least common ancestors in the schema tree. For example, consider the schema tree and assume the set of attributes in the query, \( \mathcal{A} \), consists of \{F,G,H\}. The XQuery variables \( \$F \), \( \$G \), and \( \$H \) are declared using auxiliary variables \( \$D \), and \( \$E \) as follows:

\[
\text{for } \$D \text{ in doc(...)//D,} \\
\text{\$F \text{ in } \$D/F,} \\
\text{\$E \text{ in } \$D/E,} \\
\text{\$G \text{ in } \$E/G,} \\
\text{\$H \text{ in } \$E/H}
\]

Note that these declarations restrict \( \$G, \$H \) pairs to be descendents of the same \( \$E \) instance. These pairs are “related” by the hierarchical representation. Similarly, \( \$F, \$G, \$H \) triplets are restricted to be descendents of their lca instance, \( \$D \).

Given a user query \( Q \), and the schema tree \( T \), the following algorithm generates the XQuery declarations. Let \( \mathcal{A} \) be the set of attributes that appear in \( Q \), and \( N_A \) be the corresponding set of nodes in \( T \). The algorithm uses a priority queue of entries, representing schema tree nodes arranged according to their level-number in \( T \). Initially, the priority queue \( P \) contains entries that correspond to the set of nodes in \( N_A \):

\[
P = \{ (n, l, -) \mid n \in N_A, \text{ and } l \text{ is the level number of } n \}
\]

The third parameter in each entry, initially null, is used for keeping the path for variable declaration. Its purpose becomes clear from the algorithm below.

**Algorithm 4.1 (XQuery variable declaration generator)**

- For each entry \( (n, l, p) \) in the priority queue \( P \) that has the lowest level (largest \( l \)), replace \( (n, l, p) \) with \((\text{parent}, l - 1, n/p)\), where \( \text{parent} \) is the parent of node \( n \) in the schema tree \( T \). Note that \( n/p \) is the path formed by adding \( n \) to the beginning of the path \( p \). The algorithm guarantees that the first node of path \( p \) is a child of \( n \).

- If a node \( m \) is listed more than once in the priority queue \( P \), we have found a least common ancestor. Note that \( m \) must be at the lowest level (corresponding to level \( l - 1 \) in the previous step). (1) For each occurrence \( (m, k, p) \) of a repeated node \( m \), generate the declaration (for the descendant corresponding to path \( p \)) as follows: let \( O \) be the last node of the path \( p \).

\[
\text{for } \$O \text{ in } \$M/p
\]

Hence, all such variables as \( \$O \) are declared using \( \$M \), the variable corresponding to their least common ancestor \( m \). The declaration for \( \$M \) itself will be added at a later iteration.
(2) Replace all entries corresponding to a repeated node \( m \) (such as \( (m, k, p) \)) by a single entry \( (m, k, -) \) in the priority queue.

- Repeat the previous steps as long as the priority queue has more than one entry.
- When \( P \) has only one entry, it has to be of the form \( (m, k, -) \). Add the declaration

  for \( M \) in doc(...)//m
  and remove the last entry from \( P \)

Adjustments

Certain adjustments are needed if the schema of the document was modified by the DTD transformation algorithm, or if the DTD contained “grouped subelements.”

- The document tree \( T \) may contain multiple copies of some nodes. Each copy of the node was given a new label to distinguish it from other copies. The path expressions generated by the XQuery variable declaration generator algorithm 4.1 may contain these new labels. Restore these labels to their original label. Further, a path expression of the form \( ...//label \) for a new label should be replaced with a path expression involving ancestors of \( label \) to uniquely identify the node. For example, \( ...//fname \) (See Example 3.1 for schema) should be replace by \( ...//faculty/name \).

- If the document schema contained “grouped subelements”, then we first modify the schema by introducing new virtual elements and eliminating grouped subelements (See Section 3.1.1). In this case the declarations generated by Algorithm 4.1 may contain virtual nodes (corresponding to virtual elements) in the path expressions. The adjustment process in this case is more involved and involves removal of virtual nodes and using \( sibling \) axis in the path expressions. We present an example to demonstrate the adjustment process in this case.

Example 4.1 (Grouped subelements) Consider the schema fragment

```xml
<!ELEMENT dept (dname, (id, sname)*)>
```

from Section 3.1.1. The following query lists students in the “CS” department:

```sql
select id, name
from HICT
where dname = 'CS'
```

Let’s assume we have used the grouping element (virtual) id-name-group to modify the schema. The translation algorithm, using the new schema, generates:

```sql
for $X in doc(...)//dept
$Y in $X/id-name-group
$Z in $Y/id
$W in $Y/name
where $X/dname = 'CS'
return {$Z} {$W}
```

The adjustment process removes the declaration involving the virtual element id-name-group and rewrites the declarations as

```sql
for $X in doc(...)//dept
$Z in $X/id
$W in $Z/following-sibling::name
```

The rest of the query is unchanged.

4.1.2 Translating queries on LICT

In this section, we present two algorithms for generating XQuery variable declarations for queries involving an LICT. The first one is simpler and can be used in certain frequent situations. The second algorithm is general. Let \( Q \) be a user query of the following form

```sql
select target-list
from LICT(n1,n2)
where conditions
```

The translation algorithm involves the generation of variable declarations and translation of where and select clauses of the SQL query into the where and return clauses of XQuery. The latter steps are the same as the translation algorithm for queries on HICT (Section 4.1.1). Here we only present algorithms for the generation of variable declarations.

Let the link be \( (n_1,n_2) \) (corresponding to the IDREF(S) and parent-of-ID nodes). In addition to subtrees corresponding to a link \( T_1 \) and \( T_2 \) (Definition 3.1), we define one more tree, \( T_3 \), as follows. Recall that the root of \( T_1 \) is the node \( n'_1 \), namely the child of the least common ancestor of \( n_1 \) and \( n_2 \) that is on the path from this lca to \( n_1 \). Add an edge from IDREF(S) node \( n_1 \) to the parent-of-ID node \( n_2 \) in the schema tree. We define \( T_3 \) as the tree rooted at \( n'_1 \) (with the new \( (n_1,n_2) \) edge added). Note that \( T_3 \) contains the subtree rooted at \( n'_2 \) because of the added link edge \( (n_1,n_2) \).

Let \( Q \) be the user query above, and \( A \) be the set of attributes that appear in \( Q \) (in its select and where clauses). If \( A \) contains an attribute that corresponds to a node that is not in \( T_1 \cup T_2 \), then issue an error message. This kind
of query needs (the join of) LICT and HICT. Further, there are multiple ways of joining LICT with HICT, and, in the absence of an explicit join condition, the system cannot automatically generate the correct query. In the following, we assume that \( A \) contains only attributes corresponding to nodes of \( T_1 \cup T_2 \).

**Special Case**

If all attributes of \( A \) correspond to nodes of \( T_3 \), the following algorithm can be used to generate variable declarations for the XQuery statement.

**Algorithm 4.2**

- Apply Algorithm 4.1 to attributes \( A \) with respect to \( T_3 \) (instead of the schema tree \( T \)) to generate XQuery variable declarations.
- Replace path expressions involving the link \( n_1/n_2 \), namely path expressions of the form \( p_1/n_1/n_2/p_2 \), where \( p_1 \) and \( p_2 \) are path expressions, with the appropriate dereferencing path expression \( p_1/n_1\rightarrow n_2/p_2 \).

We demonstrate this case with an example.

**Example 4.2** Consider the schema of Example 3.4. The schema tree is

```
dept
  faculty
    name
    ra
    ta
  student
    sid
    sname
```

The query to list faculty and their ra's is:

```
select fname, sname
from LICT(ra-student)
```

It is easy to verify that the set of attributes in the query correspond to (subset of) nodes of \( T_3 \). The algorithm generates the following declarations, and XQuery:

```
for $FAC in doc(...)//faculty
  $FN in $FAC/fname
  $SN in $FAC/@ra->student/sname
return {$FN} {$SN}
```

**General Algorithm**

In this case the set of query attributes \( A \) includes attributes outside \( T_3 \). Recall that we are assuming \( A \) only involves attributes corresponding to the nodes of \( T_1 \cup T_2 \).

Let \( A_1 \) and \( A_2 \) be the set of query attributes that correspond to nodes of \( T_1 \) and \( T_2 \), respectively.

**Algorithm 4.3**

- Add the IDREF node \( n_1 \) to \( A_1 \), and execute algorithm 4.1 on these nodes with respect to \( T_3 \). This will generate variable declarations for variables corresponding to \( A_1 \) plus \( n_1 \). Let the variable corresponding to \( n_1 \) be \( $N1 \).
- Add the parent-of-ID node \( n_2 \) to \( A_2 \), and execute algorithm 4.1 on these nodes with respect to \( T_2 \). This will generate variable declarations for variables corresponding to \( A_2 \) plus \( n_2 \). Let the variable corresponding to \( n_2 \) be \( $N2 \).
- Add the following condition to the where clause of XQuery: \( $N2 = id($N1) \).

4.1.3 Translating queries involving multiple HICTs and/or LICTs

When the user query involves multiple HICT/LICT in its from clause, we can translate the query by using Algorithms 4.1, 4.2, and 4.3 to generate XQuery variable declarations for each HICT or LICT separately, and translate the select and where clauses as before. In some cases, the resulting XQuery can be further optimized. The following example demonstrates this process.

**Example 4.3** Using the following schema (which is a slight variation on schema of Example 3.1) the following query lists pairs of faculty in the same division.

```
<!ELEMENT dept (division*)>
<!ELEMENT division (faculty*)>
<!ELEMENT faculty (name, ra*, ta*)>
<!ELEMENT ra (name)>
<!ELEMENT ta (name)>
select h1.fname,h2.fname
from HICT h1 h2
where h1.division=h2.division
```

The translation algorithm generates:

```
for $D1 in doc(...)//division
  $F1 in $D1/faculty/fname
  $D2 in doc(...)//division
  $F2 in $D2/faculty/fname
where $D1 = $D2
return {$F1} {$F2}
```

This query can be further optimized by realizing that the condition \( $D1 = $D2 \) implies that these two variables are bound to the same element and can be used interchangeably. The optimized query is:
for $D1$ in doc(...)//division
    $F1$ in $D1/faculty/fname
    $F2$ in $D1/faculty/fname
return {$F1} {$F2}

Optimization Principle

In the document tree, each node (except the root) has a unique parent. Consider an XQuery with a where-clause condition $x1 = x2$, where $x1$ and $x2$ are variables on the same node type, and denote node identifiers (rather than node values). We can infer that (1) $x1$ and $x2$ refer to exactly the same node, and can be used interchangeably, and (2) If $y1$ is the k-th ancestor of $x1$ and $y2$ is the k-th ancestor of $x2$, then $y1 = y2$ also holds. These observations can be used in optimization of the XQuery obtained by the translation algorithms. The example above demonstrates a simple case of such optimization.

4.1.4 Translating queries involving order

Queries involving order were discussed in Section 3.4. These queries have conditions of the form \( \text{att-pos} = n \) in their where clause, where att corresponds to a node in the schema tree. The translation algorithm is the same as before, with the following modifications: (1) All attributes att such that a condition \( \text{att-pos} = n \) appreas in the where clause are included in the set of query attributes \( A \), and (2) The order condition is added to the declaration of the XQuery variable corresponding to the attribute att. The following example demonstrates the translation algorithm.

Example 4.4 (Translating queries involving order)
Consider the second query from Example 3.3 repeated here for convenience:

\begin{verbatim}
select rname
from hict
where faculty-pos=5 and ra-pos=2
\end{verbatim}

The translated XQuery is:

\begin{verbatim}
for $F$ in doc(...)//faculty[5]
    $R$ in $F/ra[2]
    $N$ in $R/rname
return {$R}
\end{verbatim}

4.1.5 Translating queries involving aggregation and group by

Translation of user queries involving group-by and aggregation is slightly more involved than regular queries. The translated XQuery has the general form

\begin{verbatim}
group-by variable declaration
variable declarations
where (translated conditions and added conditions)
return (translated select clause)
\end{verbatim}

We will present the algorithm for queries involving a single group-by attribute first.

Algorithm 4.4 (Translating aggregate queries)

1. Declaration for a variable \( G \) corresponding to the group-by attribute is generated using \( \text{distinct-values} \) function.
2. Declarations for variables corresponding to SQL query attributes are generated using a slightly modified version of Algorithm 4.1. Variables corresponding to aggregate attributes and their auxiliary variables should be declared using \( \text{let} \) (instead of \( \text{for} \)). Further, equality conditions to the group-by variable \( G \) are included (see example 4.5.)
3. Finally, the XQuery return clause is generated by a simple translation of the SQL select clause.

Example 4.5 Consider the following schema, and user query:

\begin{verbatim}
books

author
  a-name

book
  title
  publisher
  price

select a-name, count(book), sum(price)
from HICT
group by a-name
\end{verbatim}

The translation algorithm generates the following XQuery:

\begin{verbatim}
for $G$ in distinct-values(doc(...)//a-name)
    let $B := for $B2 in doc(...)//book
        where $B2/author/a-name = $G
        return ($B2)
    $P := $B/price
return {$G, count($B), sum($P)}
\end{verbatim}

The declaration for variable \( B \) can be simplified as:

\begin{verbatim}
let $B := doc(...)//book[author/a-name = $G]
\end{verbatim}

Extension to aggregate queries with multiple group by attributes

The algorithm follows the same pattern as queries with a single group-by attribute. Declaration of variables that correspond to group-by attributes needs additional detail. These declarations should enforce that an instantiation for these variables correspond to valid data in the document. For example, consider an XML document listing stores in a supermarket chain, and for each store, it contains data such as state, city, and sales figures. An aggregate query
that groups by state and city should make sure state and city value pairs are valid, i.e., the document contains a store with that state/city pair. This can be achieved, in general, by including conditions in the declarations that connects each group-by variable with all group-by variables declared previously. As an example, these declarations for our store sales example could be:

\[
\begin{align*}
&\text{for } s \in \text{distinct-values(doc(...)//state),} \\
&\quad s \in \text{distinct-values(doc(...)//store[state=}$s$/city)}
\end{align*}
\]

The rest of the translation algorithm is similar to the algorithm for queries with single group-by attribute.

4.2. Correctness of Translation Algorithms

In this section we show the correctness of our translation algorithms:

**Theorem 4.1** Let \( D \) be an XML document, and \( Q \) be an SQL user query on the ICT (HICT and LICTs) of \( D \). Then the answer to \( Q \) is the same as the answer to the translated XQuery \( Q' \) on document \( D \).

We sketch the proof for the two basic cases: Queries on (a single) HICT (Section 4.1.1), and queries on (a single) LICT (Section 4.1.2). Correctness of more general queries follow from these basic cases.

In our proof we use the following instantiation-based semantics for SQL queries. Consider a query \( Q \) on a single table \( r \). Let \( A \) be the set of attributes appearing in the query \( Q \). An instantiation is a mapping \( \nu \) that maps \( A \) to domain values appearing in \( r \), such that for some tuple \( t \in r \), \( \nu(A) = t(A) \) for all \( A \in A \). Intuitively, an SQL query is evaluated by applying the where-clause conditions to all instantiations, and generating the output (as specified by the select-clause) for the instantiations that satisfy the where-clause conditions.

We will show that, given a (SQL) user query, the set of instantiations for the attributes corresponds exactly to the set of bindings for the translated XQuery variables. Since the XQuery where and return clauses are simple rewritings of SQL where and select clauses, it follows that the answers to the SQL query and the translated XQuery are the same.

**Case 1: Queries on HICT.** We have shown that tuples of HICT correspond exactly to the set of valuations (or valid valuations for cyclic schema) (Theorems 3.2, 3.3). The algorithm to generate XQuery variable declarations, Algorithm 4.1, generates the declarations for the variables that correspond to SQL query attributes. It utilizes additional variables, that correspond to least common ancestors of these attributes, in these declarations in a way that, when the XQuery is executed against the document, the bindings for each variable pair come from their lca element. This insures that the bindings for the set of variables correspond to a valuation. It follows that the set of instantiations for SQL attributes is the same as the set of bindings for the corresponding XQuery variables. We should note here that it is possible for ICT representation to produce multiple copies of the same data. For example, in our faculty, ra, ta case (Example 3.1), a faculty can be listed as many times as the number of ra’s and ta’s he/she has. The use of (implicit or explicit) distinct removes this extra duplication to preserve multi-set equivalence of user query with the translated XQuery.

**Case 2: Queries on LICT.** Let LICT\((n_1, n_2)\) be the LICT corresponding to the link \((n_1, n_2)\), and \(T_1\) and \(T_2\) be the subtrees corresponding to \((n_1, n_2)\) link as defined in Definition 3.1. First, we make the following observation:

A tuple \( t \) in LICT\((n_1, n_2)\) has the following property. Let \( N_1 \) be the nodes of \( T_1 \) and \( N_2 \) be the nodes of \( T_2 \). Note that these nodes correspond to the attributes of LICT\((n_1, n_2)\). (1) \( t[N_1] \) is a valuation on \( T_1 \), (2) \( t[N_2] \) is a valuation on \( T_2 \), and (3) \( t(n_1) \) and \( t(n_2) \) are related by the link \((n_1, n_2)\).

Now, the translation algorithm that generates variable declarations for the XQuery, Algorithm 4.3 (the general case algorithm), enforces that the bindings to XQuery variables have the following property: (1) Variables corresponding to nodes of \( T_1 \) correspond to a valuation on \( T_1 \), (2) Variables corresponding to nodes of \( T_2 \) correspond to a valuation on \( T_2 \), (3) the additional XQuery where clause condition generated by the algorithm enforces that these valuations for \( n_1 \) and \( n_2 \) are related by the IDREF/parent-of-ID link. It follows that the instantiations for SQL attributes are the same as the binding for XQuery variables.

5. Structuring output into XML

The output of a regular SQL query is a table. In this section we will provide a simple syntactical extension to SQL, called SQL-X, that makes it possible to specify the output formatted into an XML document. We will introduce SQL-X with an example, then briefly sketch the translation algorithm.

**Example 5.1 (SQL-X)** Consider the schema of Example 3.1. The query “list faculty and their ra’s” formatted as XML with faculty elements containing fac-name and RAs subelements listing names of all ra’s,” is written in SQL-X as follows:

\[
\text{select <faculty group by faculty>}
\quad <\text{fac-name}>\text{fname}</\text{fac-name}>
\quad <\text{RAs}>
\quad <\text{ra-name}>\text{rname}</\text{ra-name}>
\quad </\text{RAs}>
\quad </\text{faculty}>
\text{from HICT}
\]

14
Example 5.2 Consider Example 1.1 again. The XML document lists stores, store information (including city, state), and monthly sales figures. Assume relationships are represented hierarchically. Consider the same query “list total annual sales for stores grouped by state, city and year”. In this example, we also want to structure the output as an XML document with the hierarchical structure state, city, year, sales. The SQL-X query is written as follows:

```sql
select <state group by state>state
    <city group by city>city
    <year group by year>year
    <annual-sales>sum(sales)</annual-sales>
</year>
</city>
</state>
from HICT
```

5.1. Translating SQL-X queries to XQuery

The translation algorithm is an extension of the translation algorithm for queries involving group-by and aggregation. We only present a sketch here. Group-by variables are generated as before. Variables at each level of nesting in the select clause should be bound to the group-by variables at that nesting level, as well as group by variables of the encompassing nesting levels. A return statement is generated for each nesting level containing tags and output variables (if any) as specified by SQL-X select clause.

6. Related work

It has been recognized for some time that one needs simpler user interfaces for querying XML databases, e.g., see [11, 2, 3, 5]. Grahne and Lakshmanan [11] point out the crucial differences between querying a semistructured database versus navigating it and argue for the need for high level declarative query languages. Bry et al. [2, 3] discuss a visual query language called Xcerpt as alternative to XQuery. Xcerpt is based on an underlying logic for querying trees based on tree patterns and thus inherits some of the advantages of the latter’s declarative semantics. However, they do not concern themselves with translation to XQuery, an important problem given the importance of XQuery as a de facto W3C standard. Besides, unlike our approach, query formulation in Xcerpt is sensitive to changes in the structure of the underlying database schema. Cohen et al. present an approach to generate relations from XML documents [5]. They characterize when a pair of nodes in the document tree can be considered to be meaningfully related, and extend this characterization to sets of nodes in terms of the graph of their pairwise meaningfully-related relationships. They argue that certain graph configurations, such as complete, star, and connected configuration, indicate that the set of nodes are related. The emphasis of the paper is on the complexity of generating the desired relation, given an appropriate specification in terms of path expressions (for each column of the output relation) and the desired graph configuration (complete, star, or connected). The choice of the graph configuration seems arbitrary: there are no obvious guidelines to determine the appropriate configuration for a given application. Further, they only study hierarchically structured documents and do not address the important case where relationships are represented by references (such as IDREFS and keyrefs). As such, their contributions are orthogonal to those of this paper. In a different context, Arenas and Libkin [1] introduce the notion of tree tuples, to which our notion of valuations are somewhat similar. However, the goals and contributions of the two papers are very different. Indeed, we see this as independent evidence that the notion of valuations arises naturally. Finally, there has considerable work on “shredding” an XML database into relations [16, 17]. The emphasis of these works is on finding efficient storage structures for XML databases using the well-developed relational technology and leveraging it for efficient query evaluation. The goal of these works is considerably different from ours.

7. Conclusions and future work

While powerful query languages such as XQuery are available for XML, query formulation against an XML database can be challenging. Furthermore, it is highly sensitive to the way data is structured (e.g., nesting structure, use of references). In this paper, we develop a simpler user interface for querying XML databases. Thereto, we captured the information content of an XML database using the notion of valuations. We provide an alternative computational semantics of the information content and show that it is equivalent to the one based on valuations. The information content can be used to reconstruct the original XML database, preserving order. Furthermore, it can be regarded as an information-preserving view of the XML database. We showed how SQL queries against the information content view can be translated to XQuery queries against the corresponding XML database. Sophisticated XQuery queries can often be formulated as simple SQL queries against the view. To facilitate flexible output construction, we discussed a lightweight extension to SQL, SQL-X, that incorporates tagging and nesting. Our query translation algorithm can handle SQL-X.

Several issues remain open. In this paper, we did not discuss DTDs with disjunction, although we have prelimi-
nary results. Further work is needed for optimizing XQuery statements generated by the translation algorithm. How do integrity constraints on the XML database (e.g., see [6]) impact the information content and query translation and optimization? Can information content be used as a basis for understanding the problem of schema equivalence for XML database schemas?

References


