Abstract—Extensible Markup Language (XML) [29], [1] is emerging as the data format of the web era. XML has the characteristics, such as the extensibility and self-description, which make it becomes a criterion of data exchanging between applications [9]. In recent years, XML has become the standard format of data representation and exchange in the scientific research and business application. For taking advantage of the potential of XML, effective storage of XML-format data is a technology track that must be solved [16]. This is creating a new set of data management requirements involving XML, such as the need to store and query XML documents. At present, three common technologies can be used to store and retrieve XML documents which are native XML database, Object Oriented Database (OODB), and Relational Database (RDB) [6]. Relational database is possessed of perfectly theory and technology support, and it has well market-based. Using it to store XML data is relatively simple and feasible. Building a correct structure and data mapping relational between them is an issue that needs to be resolved for XML document [16]. The wide variety of XML applications and the mismatch between XML’s nested-tree structure and the flat tuples of the relational model make storing XML documents in relational databases as interesting challenges. The problem of transforming XML data from XML files into the tabular data that can be managed by relational database management system is a very interesting problem [21], [16]. In this paper, we present a survey about the recent and previous mapping approaches that map XML documents to RDB. This survey will support the future research and development work as well as to raise the awareness for the presented approaches. In this survey, we study how the various definitions in a given XML DTD, such as elements, attributes, parent-child relationships, and ID-IDREF(s) attributes can be mapped to entities and relationships. How algorithms which map DTD to relational schema and as well as content and structure preserve the functional dependencies during the mapping process in order to produce relations with less redundancy. Finally, How the recent and previous approaches translate XML query languages, such as XQuery and xpath languages into SQL.

Keywords: XML DTD, XQuery, Relational Database, SQL

I. INTRODUCTION

XML is an emerging standard for the representation and exchange data over the internet. XML is a subset of SGML (Standard Generalized Markup Language). As such, XML tags allow to describe the meaning of the content itself. New tags and attribute names can be defined, document structures can be nested to many levels of complexity, and documents can be associated with a type specification called document type definition (DTD). Relational databases are particularly good for storing and querying highly structured information. As a query language, SQL is designed specifically to query structured data. In addition, RDBMSs store data efficiently and with no redundancy because each unit of information is saved at only one place. RDBMSs are also known for their reliability and scalability and such systems can be accessed by a very large number of concurrent users. Relational systems were never designed to handle semi-structured content often stored as XML. Semi-structured is often explained as schemaless or self-describing terms that indicate that there is no separate description of the type or structure of data. Semi-structured content is difficult to store in relational database since it does not map easily to the row-and-column structure of a RDBMS. Hence there exists a need for a technique to transform data from XML to Relational tables. The rest of this paper is organized as follows. Section 2 describes the different techniques in mapping a given XML schema to relational schema. The mapping from XQUERY to SQL is described in Section 3. The conclusion is shown in Section 4.

II. MAPPING XML DOCUMENTS TO RDBMS

In this section, we present a survey about the recent and previous mapping techniques that describe how the various definitions in a given XML DTD, such as elements, attributes, parent-child relationships, and ID-IDREF(s) attributes can be mapped to entities and relationships. How algorithms which map DTD to relational schema and as well as content and structure preserve the functional dependencies during the mapping process in order to produce relations with less redundancy.

A. Storing XML documents and XML policies in Relational Databases

In [7], we have described our technique to map a given XML DTD to a relational schema. Our technique has consisted of eight steps:

1) transform the DTD to Xschema.
2) Simplify the Xschema constraints.
3) inlining of elements and attributes.
4) handling key constraints.
5) mapping collection types.
6) mapping IDREF and IDREFS attributes, (IDREFS attributes are treated similar to child elements).
7) handling the union type(or).
8) capturing the order specified in the XML model.

1) Transforming DTD to Xschema: First we define XSchema, a language independent formalism to specify XML schemas[18]. To define XSchema, we first assume the existence of a set $E$ of element names, a set $A$ of attribute names and a set $\hat{\tau}$ of atomic data types (e.g., ID, IDREF, IDREFS, string, integer, date, etc). Xschema is a structural specification of an XML schema with specification of data types, attribute definitions, and inclusion dependency constraints. Further attributes of types IDREF and IDREFS identify the target types referred to by the values.

**Definition 1.** An XSchema is denoted by 5-tuple $X=(E, A, M, P, r)$, where:

- $E$ is a finite set of element names in $\hat{E}$,
- $A$ is a function from an element name $e \in E$ to a set of attribute names $a \in A$,
- $M$ is a function from an element name $e \in E$ to its element type definition: $M(e) = \alpha$, where $\alpha$ is a regular expression: $\alpha ::= \epsilon | \tau | \alpha \cup \alpha | \alpha \vec{\alpha}$, where $\epsilon$ denotes the empty element, $\tau \in \hat{\tau}$, $\vec{\tau}$ for the concatenation, $\alpha \vec{\alpha}$ for the Kleene star, $\vec{\alpha}$ for $(\alpha \cup \epsilon)$,
- $P$ is a function from an attribute name $a \in A$ to its attribute type definition: $P(a) = \beta$, where $\beta$ is a 4-tuple $(\tau, n, d, f)$, where $\tau \in \hat{\tau}$, $n$ is either $\epsilon$ (nullable) or $\neg \epsilon$ (not nullable), $d$ is a finite set of valid domain values of $\alpha$ or $\epsilon$ if not known, and $f$ is a default value of $\alpha$ or $\epsilon$ if not known. Further more, if $\tau$ is IDREF or IDREFS, then $\tau$ also specifies the target type or types that the attribute value should refer to using the symbol $\vec{\tau}$,
- $r \subseteq E$ is a finite set of root elements.

**Example 1.** The following is the DTD for a Conference:

```xml
<!DOCTYPE Conference [
  <!ELEMENT conf (title, date, editor)"apers")>
  <!ATTLIST conf id ID # REQUIRED>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT date EMPTY>
  <!ATTLIST date year CDATA # REQUIRED>
  mon CDATA # IMPLIED>
  <!ELEMENT editor (person"
  <!ATTLIST editor eids IDREFS # IMPLIED>)
  <!ATTLET person (name, email, phone")
  <!ELEMENT person name, email, phone")
  <!ATTLET person id ID # REQUIRED>)
  <!ELEMENT name EMPTY>
  <!ATTLET name fn CDATA # IMPLIED>
  ln CDATA # REQUIRED>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT phone (#PCDATA)>
  <!ELEMENT cite (papers")>
  <!ATTLET cite id ID #REQUIRED format (ACM/IEEE) #IMPLIED>
  ]]
```

Fig. 1. Example of XML document

2) Simplify the Xschema constraints: Since the relational model cannot capture all the constraints specified in the XSchema, then we try to simplify the XSchema to transform it to relational schema. Our schema simplification step is based on the following principles[23, [4]:

$$(e1, e2)^* \rightarrow e1^*, e2^*$$

$$e1^* \rightarrow e1^*$$

$$e2^* \rightarrow e2^*$$

Where $e1, e2$ and $a$ are subelements.

**Example 2.** In the above XSchema $M(person) = (name, email, phone")$ is simplified to $M(person) = (name, email", phone")$.

3) Inlining: Our inlining technique creates one relation for an element instead of creating more relations corresponding to one element which is performed in [18]. Inlining is used to generate more meaningful and efficient relational schemas. In inlining, we consider attributes of descendants of an element as attributes in the relation corresponding to that element. Inlining for an element (e) is done recursively using the in-line technique described below. Inline technique returns a relation that should be generated for an input element currEl. The inlining technique also takes as input attSet which is used to maintain the list of attributes of (e) that should be present in the relation generated for (e). To inline the element (e), we call inline, where the initialization is: currEl = e, attSet = $\phi$. inline : currEl, attSet -> ResultSet

1. Assign the set of attributes in A(currEl) except IDREF and IDREFS attributes to attSet.
2. Set ResultSet = $\phi$.
3. Let the elements which occurs in M(currEl) with occurrence

$$^\epsilon$$

$^\tau$
4. Since $M(\text{author}) = \{\text{person} \text{ in paper}\}$ element, then we add a foreign key to the paper table in addition to $\text{paper} \text{ table}$ will be $\text{person}(\text{code, id, fn, ln, email, phone, conf, paper})$ and paper table will be $\text{paper}(\text{code, id, title, contact, cite\_id, cite\_format, conf, paper})$.

5. Return ResultSet.

The inlining technique is applied to the top elements which are determined by the following rules[19]:

Rule1: An element which does not appear in any other element type definition (such as $\text{conf}$).

Rule2: A non#$\text{PCDATA}$ element which appears in more than one other element type definition.

Rule3: A non#$\text{PCDATA}$ element B which appears in another element type definition A with "$\ast\ast$" or "$\ast$" operators (such as paper, person).

Rule4: If recursion occurs, one of the elements in the recursion is selected as a top element.

Example 3. According to the above rules we find that the element nodes are $\text{conf}$ (according to rule 1), $\text{paper}$ (according to rule 3, 4) and $\text{person}$ (according to rule 3), so by performing inlining on $\text{conf}$, $\text{paper}$, and $\text{person}$, we obtain the following relation definitions $\text{conf}(\text{id, title, year, mon, day, editor})$, $\text{paper}(\text{id, title, contact, author, cite\_id, cite\_format})$, $\text{person}(\text{code, id, fn, ln, email, phone})$.

4) Handling key constraint: In each relation that is created from the previous step, add an attribute (code), its values are $1,2,3...,\text{etc.}$, as a primary key for each relation.

Example 4. The relation definitions will be $\text{conf}(\text{code, id, title, year, mon, day, editor})$, $\text{paper}(\text{code, id, title, contact, author, cite\_id, cite\_format})$, $\text{person}(\text{code, id, fn, ln, email, phone})$.

5) Mapping collection types: 1. If there is a table corresponding to the collection type, then adds a foreign key refers to the table that represents or contains its parent.
2. Else create a new table corresponding to the collection type, and add a foreign key refers to the table that represents or contains its parent.
3. If the parent of the collection type say (e) is an attribute in a table and $A(e)=\Phi$, then remove it form that table.

Example 5.
1. Since $M(\text{conf})=\{\text{title, date, editor }\text{ paper}\}$, and there is a table corresponding to $\text{paper}$ element, then we add a foreign key to the $\text{conf}$ table in $\text{paper}$ table. So the $\text{paper}$ table will be $\text{paper}(\text{code, id, title, contact, author, cite\_id, cite\_format, conf})$.
2. Since $M(\text{editor})=\{\text{person}\}$, and there is a table corresponding to $\text{person}$ element, then we add a foreign key to the $\text{conf}$ table in $\text{person}$ table. So the $\text{person}$ table will be $\text{person}(\text{code, id, fn, ln, email, phone, conf})$.
3. Since $M(\text{cite})=\{\text{paper}\}$, and there is a table corresponding to $\text{paper}$ element, then we add a foreign key to the $\text{paper}$ table in $\text{paper}$ table. So the $\text{paper}$ table will be $\text{paper}(\text{code, id, title, contact, author, cite\_id, cite\_format, conf, paper})$.
4. Since $M(\text{author})=\{\text{person}\}$, and there is a table corresponding to $\text{person}$ element, then we add a foreign key to the $\text{paper}$ table in $\text{person}$ table and remove the author attribute form $\text{paper}$ table. So the $\text{person}$ table will be $\text{person}(\text{code, id, fn, ln, email, phone, conf, paper})$ and paper table will be $\text{paper}(\text{code, id, title, contact, cite\_id, cite\_format, conf, paper})$.

6) Mapping IDREF and IDREFS attributes: 1. An IDREF attribute is mapped by replacing it by a foreign key.

Example 6. We have an IDREF attribute defined for contact, which refers to $\text{person}$. So the result of our mapping is defining the contact attribute in $\text{paper}$ table as a foreign key refers to $\text{person}$ table.
2. IDREFS attributes are mapped by creating a new table contains a foreign key to the referenced table and a foreign key for the table that represent the element which contains the IDREFS attribute, then removing the IDREFS attribute form that table.

Example 7. We have $\text{conf}(\text{code, id, title, year, mon, day, editor})$, $\text{paper}(\text{code, id, title, contact, author, cite\_id, cite\_format})$, $\text{person}(\text{code, id, fn, ln, email, phone, conf})$.

So we create a new table $\text{editor}(\text{conf, person})$ and remove the editor attribute form $\text{conf}$ table, so $\text{conf}$ table will be $\text{conf}(\text{code, id, title, year, mon, day})$.

7) Handling the union type: In this step, if we find more than one attribute that may be NULL in a table say $a1$ and $a2$, do the following step:
1. Replace them with two attributes, one of them as a flag attribute and the second for values its name is $\text{a1}_\ast\_\text{a2}$.
2. Add the flag attribute to the key of the table.

Example 8. We have $\text{person}(\text{code, id, fn, ln, email, phone, conf, paper})$, since email and phone attributes may be NULL, so we replace them with a flag attribute that added to the key and email\_phone attribute, then we will have $\text{person}(\text{code, flag, id, fn, ln, email\_phone, conf, paper})$.

8) Capturing order specified in the XML model: To capture document order in relational database system, we encode each element’s position in an XML document as a data value by using Dewey order method[28]. With Dewey order, each element is assigned a vector that represents the path from the document’s root to the element. We store the order of elements in XML DTD in two tables, these tables are meta data that will be used in mapping relational query result to XML documents.

Example 9. For DTD conference, we create the following tables:
A. A. Abd El-Aziz et.al.  

To relational that takes into account the integrity constraints of XML Schema to Relational


is what they have called normalized relations for normalized it, every relation generated by algorithm 1 is in BCNF. That is normalized according to the functional dependencies over relations. Finally the have proved that if the original DTD has been normalized, the generated relations will be in BCNF. So their method has preserved the good properties of normalized DTD, and can fully leverage the relational technology. The mapping algorithm is shown below:

then Zijing et al., have defined the redundancies and normalization for a given DTD, and have investigated the relationship between FDs in original DTD and those in the generated relations. Finally the have proved that if the original DTD is normalized according to the functional dependencies over it, every relation generated by algorithm 1 is in BCNF. That is what they have called normalized relations for normalized XML document.

C. XShreX: Maintaining Integrity Constraints in the Mapping XML Schema to Relational

In [15], Qiuju Lee et al., have proposed a mapping of XML to relational that takes into account the integrity constraints expressed in XML Schema. they have presented an extension, called XShreX, of the ShreX mapping [5]. they have reported the preliminary results of a comparative performance analysis using a mainstream commercial relational database management system. The results have suggested that the extension of ShreX does not come at a prohibitive cost for insertions, deletions, updates and queries. In the case of queries, XShreX can even yield a performance improvement.

1) ShreX Mapping Approach: By default, ShreX maps all XML Schema elements to relational fields with NOT NULL constraints, unless their minOccurs attribute is set to 0. It inlines simpleType elements and attributes within their parent elements tables. As attributes are optional in XML Schema, they are not mapped with NOT NULL constraints. However a required attribute will be mapped with a NOT NULL constraint. ShreX maps complexType and repetition elements to a separate table from their parent elements tables. ShreX generates a list of auto-generated numeric primary and the corresponding foreign keys. The child table references the primary key of the parent table. choice elements are either simpleType or complexType. simpleType elements are inlined within their parent elements table. complexType elements are mapped to a separate table from their parent elements tables. ShreX provides an interface to define data storage and mapping, the proposed XShreX has aimed to enrich this tool with additional constraints mapping.

2) Not Null constraint: The not null constraint specifies that the element must not contain a null value. Its minOccurs attribute is set to at least 1. This constraint can be expressed using ShreXs existing rules. By default, ShreX sets all fields to NOT NULL in the relational tables.

3) Cardinality constraints. Cardinality: constraints determine the occurrence of an element in XML Schema. These constraints can be expressed using ShreXs existing rules. (0,1) cardinality indicates that the element is optional. Its minOccurs attribute is set to 0. The mapped relational field will not be set as the default NOT NULL. (1,1) cardinality indicates that the element is compulsory. The mapped relational field will be set as the default NOT NULL. (X,N) cardinality indicates that the element can appear repeatedly for X up to N occurrences. Its maxOccurs and maxOccurs attributes are set to X and

<table>
<thead>
<tr>
<th>pathId</th>
<th>ele_name</th>
<th>parentId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>conf</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>title</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>date</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>editor</td>
<td>1</td>
</tr>
<tr>
<td>1.3.1</td>
<td>person</td>
<td>1.3</td>
</tr>
<tr>
<td>1.4</td>
<td>paper</td>
<td>1</td>
</tr>
<tr>
<td>1.4.1</td>
<td>title</td>
<td>1.4</td>
</tr>
<tr>
<td>1.4.2</td>
<td>contact</td>
<td>1.4</td>
</tr>
<tr>
<td>1.4.3</td>
<td>author</td>
<td>1.4</td>
</tr>
<tr>
<td>1.4.4</td>
<td>cite</td>
<td>1.4</td>
</tr>
<tr>
<td>1.4.3.1</td>
<td>person</td>
<td>1.4.3</td>
</tr>
<tr>
<td>1.4.4.1</td>
<td>paper</td>
<td>1.4.4</td>
</tr>
</tbody>
</table>

**TABLE I**

**THE DEWEY ORDER OF THE CONFERENCE XML DOCUMENT**

<table>
<thead>
<tr>
<th>parent</th>
<th>sub_ele</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>person</td>
<td>name</td>
<td>1</td>
</tr>
<tr>
<td>person</td>
<td>email</td>
<td>2</td>
</tr>
<tr>
<td>person</td>
<td>phone</td>
<td>3</td>
</tr>
</tbody>
</table>

B. Storing Normalized XML Documents in Normalized Relations

In [27], Zijing et al., have studied the XML storage in relations. Unlike traditional techniques, they have considered the semantics expressed by functional dependencies. They have proposed an algorithm for mapping DTD to relational schema, which preserves not only the content and structure but also the semantics of original XML documents. To tackle the problem of constraints expression, they have introduced a way to define functional dependencies and normalization for DTD. In a normalized DTD, every constraints have been expressed by functional dependencies can be concluded to keys. So they have used the key definitions for XML as the foundation for relation generation and maintain the keys in relations. After investigating the relationship between functional dependencies in XML documents with the corresponding ones in relations, they have proved that, if the original DTD has been normalized, the generated relations will be in BCNF. ShreX provides an interface to define data storage and mapping, the proposed XShreX has aimed to enrich this tool with additional constraints mapping.

Algorithm 1: Mapping DTD to Relations

1. Create a DTD graph to represent the structure of given DTD, including elements, attributes and operators. Also add the virtual root, and split the shared nodes if necessary.
2. Create key relations for the chosen set of keys from original DTD (described in the previous subsection). If K = (R1, R2, Q1, ... , Qn), mark the node for last (R2) in DTD graph.
3. Define (K1, K2, K3, ..., Kn) and (K1 ID, K2 ID, K3 ID, ..., Kn ID) as keys for key relation K1, K2, K3, ..., Kn.
4. Identify top nodes that need a separate relation.
5. Start from top node T, define all the element and attribute nodes that are reachable from T unless they are other top nodes.
6. Add a XID field as key for all the generated relations other than key relations.
7. Add a parent ID field for relations to record the key value of parent element if necessary, and if the the parent element X is inlined into another element Y, record the key value for Y instead.
N respectively. The element will be mapped as a repetition element to a separate table from their parent elements tables.

4) Structural sequence constraint: Structural constraints preserve the structure and order of elements in a XML document. For the structural sequence constraint, the sequence element specifies that its sub-elements must appear in the order they were declared. This constraint can be expressed using ShreXs existing rules. The sub-elements will be mapped to relational fields in the same order they were declared in the XML Schema.

5) Structural choice constraint: For a structural choice constraint, the choice element specifies that only one of its sub-elements can occur. simpleType sub-elements will be inlined within the choice elements parents table. A CHECK CHECK ((field1 is NOT NULL AND field2 is NULL) OR (field1 is NULL AND field2 is NOT NULL)) is set in the table to check that only one field will contain a value. complexType sub-elements are mapped respectively to separate tables. An assertion is set to check that only one table will contain a value.

6) Unique constraint: A unique constraint specifies that the value of an element must be unique among values of the same element type. The unique element expresses this constraint. The reserved keyword UNIQUE is set to the field which the element is mapped to in the relational.

7) Key and Foreign Key constraints: A key constraint specifies that the value of an element identifies it uniquely in the XML document. The key element specifies this constraint. A foreign key constraint specifies that the value of a referring element must match the value of the referenced key element. The keyref element specifies this constraint. The reserved keywords PRIMARY KEY are set to the field which the key element is mapped to in the relational table. The reserved keywords FOREIGN KEY are set to the field which the keyref element is mapped to.

8) Domain constraints: Domain constraints specify the range of values an element can have. Its default attribute specifies a default value to be assigned to the element if no other value is specified. The reserved keywords DEFAULT value is set to the field which the element is mapped to in the relational table. The fixed attribute specifies a fixed value to be assigned to the element. No other value can be assigned. A CHECK CHECK (field = value) is set to the field which the element is mapped to. The domain enumeration constraint specifies a range of values that can be assigned to an element. Restriction facets minInclusive, maxInclusive, minExclusive and maxExclusive are used. A CHECK CHECK (field lowOp lowRange AND field highOp highRange) is set to the field. which the element is mapped to. lowOp lowRange is > minExclusive or ≥ minInclusive and highOp highRange is < maxExclusive or ≤ maxInclusive.

D. Schemaless Approach of Mapping XML Document into Relational Database

In [6], a general method for mapping XML documents to RDB has been described. The method does not need a DTD or XML schema. And it can be applied as a general solution for any tree data structure and not just for XML data. Also, it can be used for data-centric and document-centric documents. Experiments on this method have showed it’s ability to maintain document structure at a low cost price and easily, building of the original document is straight forward, performing first level semantic search is achievable either on a single document or on all documents. The mapping algorithm is depicted in Figure 3.

The data model used for the mapping algorithm uses the W3C’s Document Object Model (DOM) to represent XML documents in memory before mapping them, it also uses a stack to traverse the xml document by pushing the children of each node onto stack in reverse order in order to preserve there order in the doc_structure field. Line 5 pushes the root element of the document to the stack. The do loop is used to construct the doc_structure field and to insert the XML tokens (elements and attributes) into token’s table (lines 6-28). In line 7, the top of stack is popped, if the popped element is ">" , that means all the children of the parent element were added to the database, and the “>” symbol is appended to the "struc" string (lines 8-10). If not (i.e. the popped element is a node), the element’s name and value are inserted into the database, and its id is appended to the "struc" string. If this element has attributes, all its attributes are inserted to the database and there ids are appended to the "struc" string. Lines (21-25) check if the element has children. If so, an “<” is appended to "struc" string, and “>” is pushed to the stack, and all its children are pushed to the stack but in reverse order. Line 26 checks the status of the stack, if it is empty, the do loop is terminated. After that, the "struc" string is inserted to the database (documents table). All element’s children are enclosed by angle brackets. The nested brackets differentiate between document’s levels, while using the letter ‘T’ and ‘A’ to differentiates between element’s children and attribute.
E. RIAL: Redundancy Reducing Inlining Algorithm to Map XML DTD to Relations

In [21], Amir et al., have proposed an algorithm which maps DTD to relational schema and as well as content and structure it preserves the functional dependencies during the mapping process in order to produce relations with less redundancy. This has been done by categorizing functional dependencies and introducing four rules to be applied to the relations created by the hybrid inlining algorithm according to each category. These rules will reduce redundancies by moving attributes, creating relations, introducing keys and preserving functional dependencies. The proposed method is based on the hybrid inlining algorithm[23].

1) Redundancy Reducing algorithm: In order to reduce redundancies and preserve the functional dependencies the authors have categorized the functional dependencies into four categories and they have introduced a rule for each category that will be applied to the relations created by the inlining algorithm, but before applying these rules they have introduced a rule which they have called it Rule0 and applied to all type of FDs in the set:

Rule0: for each path p ∈ Epaths(D) if P is (LHS) or (RHS) of a FD then a relation is created for the last(p) if it is not already created by the inlining algorithm, i.e., last(p) should not be inlined in other relations.

Now the introduced four rules are:

- **Rule1** (Attribute Moving): For each FD of the form q → p.@l where p.q ∈ Epaths(D), if q → p ∉ (D,Σ)+ then inline @l in the relation correspondent to last(q).

- **Rule2** (Relation Creating): For each FD of the form {q, p₁, @l₁, ..., pₙ, @lₙ} → p.@l where q ∈ Epaths(D) n≥1 create a new relation with @l₁, @l₂, ..., @lₙ as attributes and set a foreign key referring to the relation created for last(q) or the relation in which last(q) is inline and a foreign key referring to the new created relation is added to the relation created for p. The key of this relation will be composition of @l₁, ..., @lₙ.

- **Rule3** (Key Generating):
  - For each FD of the form p₁, @l₁, ..., pₙ, @lₙ → q where p ∈ Epaths(D) set p₁, @l₁, ..., pₙ, @lₙ as the primary key of the relation. We can here remove the surrogate key introduced by the inlining algorithm.
  - For each FD of the form q₁, S, ..., qₙ, S → p where p ∈ Epaths(D) and p is a proper prefix of qᵢ, 1 ≤ i ≤ n, set last(q₁), ..., last(qₙ) as the primary key of the relation. They were sure that last(qᵢ) is inline in p, because there cannot be a *-edge between last(p) and last(qᵢ).

- **Rule4** (FK/CK¹ Generating): For each FD of the form {p₁, ..., pₙ, P@l} → q where p₁, ..., pₙ ∈ Epaths(D) consider R, R₁, ..., Rₙ are relations for last(p₁), ..., last(pₙ) respectively. For each Rᵢ, 1 ≤ i ≤ n, a foreign key fᵢ is added to R referring to Rᵢ if it already is not present. The composition these foreign keys with @l will be a candidate key for R.

By mapping the DTD using the hybrid inlining algorithm the FDs like the one introduced in Rule1 are lost because the LHS and the RHS will be mapped into two different relations; so the Rule1 is to preserve these kinds of dependencies in the target relations by moving attributes. By applying Rule2 a new relation is created and by setting the correct key for this relation the FD is preserved. By applying Rule3, the relation does not need the surrogate key introduced by the inlining algorithm and the redundancy is reduced. Rule4 preserves another type of FD by introducing foreign key to the appropriate relations and making a composite key out of them. This approach will not reduce redundancy but it preserves the semantic constraints introduced by FDs. Because the set of FDs is finite and they applied the rules on the relations created by the inlining algorithm its guaranteed that the algorithm will be terminated finally.

F. Mapping XML DTD to Relational Schema

In [10], YE Feng and Xiao Jingsheng have discussed the method to achieve mapping XML DTD to Relational Schema. They have expressed the nodes, attributes and semantic constraints of XML DTD by using ADG. Then they have optimized the ADG and finally converted ADG to relational schema. They have proposed an algorithm steps of creating ADG by DTD, the optimization of ADG, and the transformation rule from semantic constraints of ADG to semantic constraints of Relational Schema. The method can not only obtain the structure information of DTD, but also maintain the semantic constraint of DTD. The method consists of three steps: creating ADG, optimizing ADG and creating relational schema.

1) Creating ADG: In [11], ADG(Annotated DTD Graph) is a DTD graph with comment. It is an improvement on DTD graph. Each element of DTD appears in ADG only once, while the times of attribute appearing in ADG is the same to the times in DTD. In ADG, G = (V, E, D), the finite set V is composed by all elements and attributes of DTD, the finite set E is binary relationship of V. According to DTD, the algorithm of creating ADG is as follows:

1) V = φ, E = φ.
2) Scan DTD documents, get an element and create its node v.
3) If the node v ∉ V, add the node to v, V = V ∪ {v}.
   If its parent node exists, the directed edge e made up by the node v and its parent node will be added into E. The direction of the edge e is from parent node to sub node. Then, sign the Cardinality on the directed edge e.
4) If the node v has some attributes, create node v’ for every attribute one by one and join node v’ into v, V = V ∪ {v’} and the directed edge e made up by node v and its attribute node v’ into E, E = E ∪ {e}. The direction of the edge e is from node v to its attribute node v’.
   Then, sign the Cardinality on the directed edge e.
5) Starting from node v, according to the depth traversal algorithm, traverse all the sub element nodes of the node v using step 3 and step 4.
6) If there are elements in DTD which are not traversed, select one of the elements and create node v adopting step 3, step 4 and step 5.
7) Repeat step 6 until all the element nodes have been traversed.

2) Optimization of ADG: Before mapping ADG to relational schema, the ADG needs to be optimized in order to reduce its complexity. Its rules of optimization[33] are:
   1) If the Indegree of a leaf node is n, n > 1, then divide the node into n nodes.
   2) If a leaf node is an element node and meet the following conditions: (a) the indegree of the node is equal to 1, and (b) the cardinality of node is (1, 1) or (1, n), then these nodes will be marked as attribute nodes.
   3) If a node meets the following conditions: (a) the cardinality of the element is (1,1) or (0,1), (b) its the Cardinal number of sub-elements is (1,1) or (0,1), and (c) the Indegree of the element is 1, then these nodes will be merged to sub nodes.
   4) If XML DTD has the constraint of fixed or enumeration value, then the SQL constraint statement is "Create Table S (...Foreign Key (w') References R (v')) ..."
   5) If XML DTD has the constraint of default values, the SQL constraint statement is "... default ..."
   6) If XML DTD has the constraint of 'OR' expressing choice, such as "A = B", using "... Check ((A is NOT NULL And B is NULL) OR (A is NULL And B is NOT NULL)) ".
   7) If XML DTD has the constraint of IDREF (S), the constraint can not directly obtain from the XML DTD and need to obtain constraint from the corresponding XML document instance. Then use the "Foreign Key" and "References" to deal with.

G. Storing DTD-Independent XML Data in Relational Database

In [14], Zin Mar Kyu and Thi Thi Soe Nyunt have presented how to create dynamic relational schema and store XML data only depend on input XML document. For this process, they have proposed DTD-independent schema mapping algorithm and data mapping algorithm. Their DTD-independent schema mapping algorithm has used data extraction approach that extracts table names and attributes of the input XML document to create a document schema and data mapping algorithm stores all extracted XML information into relational tables by mapping outputs of DTD-independent schema mapping algorithm. The proposed system has reduced the step of creating relational schema depend on DTD information. So we do not need DTD information, can solve the problem of missing elements and attributes of inputs DTD and XML document, and will get similar result of DTD-dependent approach.

Firstly, the authors have to construct a document schema to map elements and its parent into the same table. Then they have to store extracted XML data in RDB. When the original XML document is required, the document can be constructed by assembling all the relevant information items from the database.

1) DTD-Independent Schema Mapping Algorithm:
   - **Step 1:** they have to extract XML data from XML document to store in RDB. Firstly, they have parsed XML document by SAX parser and extracted start element, start elements attributes and characters of the XML document. In this step they have gotten elementList and attributeList by recording start elements and characters. The outputs of this step are the information to store in RDB.
- **Step 2**: In order to extract table name, they should have distinct elements. So in this step they have gotten elementList1 by reducing multiple copies of same element in the elementList of step1 with the same parents into single element. Elements can separate tables if these elements are not leaves. Thus they also have gotten tableNameList by extracting distinct elements parent names of elementList.

- **Step 3**: And then, if elementList1 of the step2 has the same element with different parent elements, then replace element name by 'parent_type' and element name, and add element name into table name list if this element name doesn’t already exist in table name list. If the element name is already exist in the table name list, replace element name by 'parent_type'. In this step they have gotten elementList2 and tableNameList as outputs.

- **Step 4**: Then, they have gotten tableElementList by reducing elements if elements in the elementList2 of step3 have the same parent and they are table name and get tableAttributeList by extracting attribute name with different parent name from the attributeList of step1. they also have added elementCount into tableNameList by counting the number of elements in tableElementList for each table name.

- **Step 5**: After finished above four steps, this is ready to create a relational schema by mapping element in the tableNameList, tableElementList and tableAttributeList of step4. In this step they have defined each of the table ID as primary key.

2) **Data Mapping Algorithm**: Data mapping algorithm that depicted in Figure 4 provides to store extracted XML data into relational database. This algorithm insert element in the elementList into tables. If element in the elementList is equal table name then add Name, parentID, and endID of element into tuple of this table. If this element’s parentID is equal parentID of the attributeList, then add Value of the attributeList into tuple. Add element’s parentName into tuple if element is equal to "parent-type" element. When the Value of element in the elementList is not equal to "null", there are two possible conditions. The first condition is this element is not "parent-type". In this case we add ID, endID and Value of element into tuple. Otherwise, we add Value of element into tuple. These processes are for each element of elementList. These processes are iterated for all elements in the elementList.

H. Mapping and Labeling XML Data For Dynamic Update

In [26], S. Subramaniam et al., have scrutinized and studied the existing approaches in terms of the table schema, complexity and support for dynamic update. In addition, they have suggested a mapping approach to map XML data into relational database, s-XML based on the persistent labeling scheme.

1) **Persistent Labeling Scheme**: Persistent Labeling Scheme was proposed by Alban Gabillon [25], an efficient and effective labeling scheme which does not require the nodes to be re-numbered when dynamic updates take place. Each node will be labeled as (1, [n, d]), where:

- l is the level of the node in the tree
- n, d is the local label of the node. Pair of (n, d) represents the n/d rationals.
- np, dp is the local label of the parent node. Given a level, local labels of the nodes are unique.
- Root element will be labeled as (0, [1, 1]) where 0 represents the level and (1, 1) represents the local label of the node. This element does not have a parent label since the node is the origin of the XML tree.

2) **XML Mapping Technique**: The mapping technique that they have proposed is based on two tables which are the ParentTable and ChildTable. The ParentTable stores the information about the parent nodes (except for the leaf nodes). The ChildLabel stores the information about the leaf nodes. The schemas for these two tables are as follows:

ParentTable (IdNode, LParent, SelfLabel)

ChildTable (IdNode, Level, Child, LParent)

where:

- IdNode is used to uniquely identify the nodes stored in the ParentTable.
- LParent refers to the combination of the level and the label of the parent node.
- SelfLabel indicates the local code of the node.

![Algorithm: DataMap](image)

Fig. 4. DTD mapping algorithm
I. Semantic Web Information Retrieval in XML By Mapping To RDF Schema

In [20], Soe Lai Phyue et al., have intended to develop web information extraction technique using semantic technology. This system has presented an approach to extract and integrate information from unstructured documents (e.g. HTML) and has been converted to standard format (XML) by using ontology. A mapping technique would be used to automatically generate RDF descriptions from XML documents and the system would create RDF Sitemap which would provide more efficient and effective for Information Retrieval (IR).

1) Architecture of the system: The extraction of information on semantic web is two parts which is mapping from XML Schema to ontologies that can be used to automatically generate RDF meta-data from XML content documents and user query information retrieval using RDF crawler. In the first phrase, this is done in two steps. Firstly, for each XML Schema in Web application, a mapping to the ontology has to be defined. A flexible way to map the content of one or more XML elements to the information required by the used ontology is needed. Secondly, for each XML content page the rules defined in the previous step are applied to generate the RDF representation. The query which is enter the user is preprocessing and crawl using the RDF crawler.

2) Mapping XML Schema to Ontologies: For mapping, the XML schema which is structure of a valid XML document and the elements being used that information can be used to define XPath expression to select elements or attributes from an XML document. Once an element! attribute is selected, its content is mapped to a position in an RDF triple which is (1) a constant value such as URL reference, (2) an XPath expression, (3) the return value (4) a resource reference. The resource identifiers are defined that can be referenced later and the subject, predicate and object of the actual triples are defined. In some cases, additional information is required to select a specific element!attribute pair by an XPath expression. The mapping can be used to generate RDF description from XML documented. Figure 5 is an XML to RDF schema mapping procedure.

3) Generation the RDF Descriptions: When a Web page is queried, the corresponding XML document is fetched and the XSLT transformation is used to generate the HTML page. Second, the RDF description has to be generated and referenced from the HTML document. To generate this description all mappings defined for this document have to be executed. Therefore, the XPath expressions have to be evaluated on this document and the result is either directly used to fill a position in an RDF triple that is defined in the mapping. If the XPath expression matches multiple elements!attributes in the XML document the procedure has to be repeated for each match.

4) Crawling the RDF pages: The RDF Crawler is used to find RDF data on the web and upload it into a relational database. The urls table stores information about the urls to be searched. Any error messages generated during downloading or parsing the data. The facts table stores the usual subject, predicate, object triples .Obviously, the facts table is populated from the RDF data found. To fill the fact table, two mechanisms are used. First of all, a crawler is provided, that recursively collects the hyperlinks found on html pages. This allows you to search for RDF in a certain neighborhood. Second, the Google search engine’s feature are exploited to search for urls with a certain keyword such as RDF in the url text. Figure 6 is the procedure used as crawling algorithm.

J. Mapping from the XML Schema to the Relational Database with Functional Dependency Preserved

In [34], the authors have analyzed the sets in the data model of XML document described with SCHEMA and defines the dependency relation, value dependency relation and two kinds of fetching value relation in the described sets. According to these relations and the constraints defined by the SCHEMA, it
also creates a group of rules mapped SCHEMA to relational database. We have proved that the mapping between SCHEMA and relational database is integral and the relation after mapped conforms to the 3rd normal form. The mapping algorithm is shown below:

The mapping from SCHEMA set S to relational database D is defined as \( \Psi : S \rightarrow D \). There are six mapping rules in \( \Psi \):

- **Rule1:** If \( e_i \in E \), \( \exists j \in E \), then \( f_j : e_i \rightarrow e_j \), or \( \exists a_j \in A \), then \( f_j : e_i \rightarrow a_j \), relation \( B_i \in D \), \( \Psi : e_i \rightarrow B_i \). Then create a primary key in \( B_i \) according to the internal rules. If \( v_i \in V \), and \( f_i : v_i \rightarrow e_i \), then create a field \( c_k \) to save the data described with \( e_i \), and the data type of \( c_k \) is \( v_i \).

- **Rule2:** for \( e_i \in E \), if \( \forall \subset e_j \not\in E \), then \( f_j : e_i \rightarrow e_j \), and \( \forall a_j \not\in A \), then \( f_j : e_i \rightarrow a_j \), according to rule 3, \( \exists v_i \in V \), then \( f_i : v_i \rightarrow e_i \), is not a root element according to rule 4, so \( \exists e_k \), then \( f_j : e_k \rightarrow e_i \) and there is no transitive dependency between \( e_k \), \( e_i \). According to rule 1, \( \Psi : e_k \rightarrow B_k \), \( B_k \subseteq D \), we can create field \( c_i \), then \( \Psi : e_k \rightarrow B_k(c_i) \). The data type of \( c_i \) is \( e_i \). If \( \theta \not=\emptyset \), then \( c_i \) can be null, \( \rightarrow e_i \).

- **Rule3:** for \( e_i \in E \), if \( \forall \subset e_j \not\in E \), then \( f_j : e_i \rightarrow e_j \), and \( \forall a_j \not\in A \), then \( f_j : e_i \rightarrow a_j \), according to rule 3, \( \exists v_i \in V \), then \( f_i : v_i \rightarrow e_i \), is not a root element according to rule 4, so \( \exists e_k \), then \( f_j : e_k \rightarrow e_i \) and there is no transitive dependency between \( e_k \), \( e_i \). According to rule 1, \( \Psi : e_k \rightarrow B_k \), \( B_k \subseteq D \), we can create relation \( B_i \subseteq D \), \( \Psi : e_i \rightarrow B_i \), and create field \( c_k \) to save the data described with \( e_i \). The data type of \( c_k \) is \( e_i \), then create the foreign key \( c_k \) in \( B_i \) that is connected to \( B_k \).

- **Rule4:** If \( e_i \in E \), \( \exists e_j \in E \) then \( f_j : e_i \rightarrow e_j \), and \( f_j : e_k \rightarrow e_i \), both are not transitive dependencies, and \( \Psi : e_i \rightarrow B_i \), \( B_i \subseteq D \), if \( \theta \not=\emptyset \), then create field \( c_i \) in \( B_k \) according to rule 1, then \( \Psi : e_k \rightarrow B_k, B_k \subseteq D \), create field \( c_i \) in \( B_k \) as a primary key connecting table \( B_i \). If \( \theta \not=\emptyset \), then the field can be null.

- **Rule5:** If \( e_i \in E \), \( \exists e_j \in E \) then \( f_j : e_j \rightarrow e_i \) and \( f_j : e_k \rightarrow e_i \), both are not transitive dependencies and \( \Psi : e_i \rightarrow B_i \), \( B_i \subseteq D \), according to rule 1 \( \Psi : e_j \rightarrow B_j \), \( B_j \subseteq D \), create a correlation table \( B_{ij} \), and respective foreign keys to \( B_i \) and \( B_j \).

- **Rule6:** If \( a_j \in A \), according to rule 3, \( \exists e_i \in E \), then \( f_j : e_i \rightarrow a_j \), and according to rule 1, \( \exists v_i \in V \), then create field \( c_i \) in \( B_j \) whose data type is \( v_i \), then \( \Psi : e_i \rightarrow B_j(c_i) \). If \( \theta \not=\emptyset \), then \( c_i \) can be null.

K. A Strategic Study of Layerly Mapping XML Schema to Relational Schema

In [17], this paper mainly has studied that XML data relational storage strategy of XML Schema-based, and has given one method of layer mapping from XML Schema to Relational Database: Taking the XML Schema as input, changing it into the model map of tree structure, and then encoding the complex elements per layer in the map. Achieved Hierarchical thinking and mapping it to relational model on the basis of that, thus achieved efficient storage of XML document.

1) XML Schema document storage structure: According to the idea of layering storage and the structure of XML Schema graph, they have designed three relational table that express relational schema mapped from XML Schema and another table that stores the content of XML document, they are:

1) Level (L_Id, Label, P_Code), this table only stores the elements with child element, where L_Id expresses ID, Label expresses the name of element, P_Code expresses the prefix code of this element, and reflects the located level of this element.

2) Element(E_Id, P_Code, Label,Flag), this table stores the elements without child element, where E_Id expresses ID, P_Code corresponds to the P_Code in Table “Level”, and expresses that element-dependent, Label expresses the element’s name, flag expresses the element’s type.

3) Attribute(A_id, P_Code, Label, Values), this table stores the value of attributes, where P_Code expresses the prefix code depended on the element in Table “Level”, Label expresses the name of attributes, Values expresses the value of attributes.

4) Text (T_Id, A_id,E_Id ,Values), this table stores text value, where A_id corresponds to the ID of elements in Table “Attribute”, E_Id corresponds to the ID of elements in Table “Element”, Values expresses the value of elements, and the type of value must meets type Flag defined by Table “Element”.

L. Converting XML DTD to Database

In [8], the author has provided an algorithm of analyzing XML DTD, accessing to elements, attributes, as well as the relationship of elements contained by XML DTD. He has created a DTD graph, converted XML documents to relational schema of database, and maintained the semantic constraints contained by XML DTD. Converting XML DTD to relational schema, XML DTD semantic constraints must be considered, otherwise the relational schema will lose the original constraints.

1) Semantic constraints of XML DTD:

1) Inclusion dependencies constraints: inclusion dependencies constraints are that a value which appears in some element or attribute must be included by another one, which are similar to the reference integrity of relational schema. Inclusion dependencies constraints have two represent forms: (a) if element Y is the sub element of X,Y is included by X, (b) IDREF(S) is used in the attribute define part of DTD, but it don’t tell definitely the address of ID which “IDREF(S)” quotes and can not obtain the constraint from DTD directly. It is an unknown constraint.

2) Equality-generating dependencies constraints: Cardinality constraints of (0,1) and (1, 1) indicate that an element in the type of X has mostly a sub element in the type of Y, which are similar to function dependencies of relational schema.
3) Tuple-generating dependencies constraints: They show that tuple of some form must appear in the table. Tuple-generating dependencies constraints have two forms, one is child constraint. It shows that every parent element X has at least one sub element Y, corresponding to (1,1),(1,n) in Cardinality constraints. Another is that parent constraint shows that every sub element X has at least one parent element Y.

2) Rules of Converting XML DTD to Relational Schema:
Converting XML DTD to relational schema, element nodes will be mapped into a relational table and attribute nodes into the fields of the table. Semantic constraints contained by XML DTD are converted to that of relational schema by using the constraint statements of SQL. The main rules are as follows:

1) ID attributes are converted to primary key in the corresponding table by using SQL constraint "PRIMARY KEY".

2) The SQL constraint is "CHECK (VALUE IN)" for the constraint the constraint of fixed value or enumeration value and "default" for the constraint of default value.

3) For "OR" sub-content constraints, such as "A | B", the SQL constraint is "Check ((A IS NOT NULL AND B IS NULL) OR (A IS NULL AND B IS NOT NULL))".

4) If v is parent element and w is a sub element of v, cardinality constraints can be divided into the following three types:
   - If the cardinal constraint is (0,1) between v and its sub element w, v and w are mapped to the same relational schema R, and u is the key attribute of R, the SQL constraint statement is "Create Table R (... Unique (w), Primary Key (u) ...) ".
   - If the cardinal constraint is (1,1) between v and its sub element w, there are three SQL constraints. (a) It contains the constraint of "w is not empty". If the attribute of w is w’ in relational schema R, w’ is not allowed to be empty. The SQL constraint statement is "Create Table R (... w not NULL) ".
   - If v and w are mapped to the same relational schema R, and u is the key attribute of R, the SQL constraint statement is "Create Table R (... Unique (w), Primary Key (u) ...) ".
   - (b) If v and w are mapped to the same relational schema R and u is the key attribute of R, the SQL constraint statement is "Create Table R (... w not NULL, Primary Key (w) ...) ". If v and w are mapped to the different relational schema R and S, u is the key attribute of R, and fk_A is the foreign key of table S, the SQL constraint statement is " Create Table S (...Foreign Key (fk_A) References R (u) ...) ".

5) If XML DTD contains inclusion dependencies constraints between v and its sub element w, v and w is respectively mapping to the attribute v’ and w’ of relational schema R and S, and v’ is the primary key of R, the SQL constraint statement is "CREATE TABLE S (... FOREIGN KEY (w’) REFERENCES R (v’) ...) ", if v’ is not the primary key of R, the SQL constraint statement is " CREATE TABLE S (... (w’) CHECK w’ (IN (SELECT v’ FROM R))) ". If XML DTD has the constraint of IDREF (S), the constraint can not directly obtain from the XML DTD and need to obtain constraint from the corresponding XML document instance, using the "Foreign Key" and "References" SQL constraints deal with it.

M. summary and comparison

<table>
<thead>
<tr>
<th>paper</th>
<th>Reference</th>
<th>functional dependency</th>
<th>integrity constraints</th>
<th>schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>preserved</td>
<td>preserved</td>
<td>used</td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td>not preserved</td>
<td>preserved</td>
<td>used</td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>not preserved</td>
<td>not preserved</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>[14]</td>
<td>not preserved</td>
<td>not preserved</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>[17]</td>
<td>not preserved</td>
<td>not preserved</td>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>[8]</td>
<td>not preserved</td>
<td>preserved</td>
<td>used</td>
<td></td>
</tr>
</tbody>
</table>

III. MAPPING XQUERY TO SQL

XQuery and SQL are based on different paradigms (imperative v.s. declarative) and manipulate different data types (ordered sequences v.s. sets of tuples). In this section, We illustrate concretely some of the challenges faced by a general XQuery to SQL translator. we present an introduction about XQuery, the relationship between XQuery and the flat relation, and a survey about the Existing XQuery-to-SQL algorithms that depend on special XML-to-RDB schema-mapping algorithm and that doesn’t.

A. Introduction to XQuery

XQuery is an XML query language for retrieving information from XML documents. XQuery for XML is like SQL for relational databases. "XQuery is derived by W3C Consortium working group from an XML query language called Quilt[3] which in turn borrowed features from several other languages including XPath 1.0, XQL, XML-QL, SQL,
and OQL:” [31]. XQuery is based on the structure of XML, and it uses this structure to provide more powerful queries for the data stored in XML. XQuery is defined in terms of the XQuery 1.0 and XPath 2.0 Data Model [32] which represents the parsed structure of an XML document as an ordered and labeled tree in which nodes have identity and may be associated with simple or complex types. XQuery can be used to query XML data that has no schema at all, data has XML Schema, data has a DTD, or any other data including databases whose structure is nested similar to XML. XQuery is case-sensitive language like XML, XQuery is a functional language consists of several kinds of expressions that can be combined together with other expressions to create new expressions. There are seven types of expressions in Xquery: path expressions to select nodes in XML structure, element constructors to construct XML structures, FLWR expressions to combine and restructure information from XML documents, conditional expressions, quantified expressions, expressions involving operators and functions, and expressions that test or modify datatypes.

1) XQuery Data Model: XQuery is defined in terms of a formal data model[2]. Every input to a query is an instance of the data model, and the output of every query is an instance of the data model. The basic of the data model is the sequence. A sequence is an ordered collection of zero or more items. An item may be a node or an atomic value. An atomic value is a value of simple types that are defined in[30], such as strings, Boolean, decimals, integers, floats, doubles, and dates. A node may be: document, element, attribute, text, namespace, processing instruction, and comment[32]. A node may have other nodes as children (attributes and namespaces are not considered as children because they don’t contained by their parents, but they describe their parents) forming node hierarchies. Nodes, such as element and attribute, have names, values, or both. A value is a sequence of zero or more atomic values. Every node has a unique identifier that distinguishes it from other nodes (in its document or in other documents) because we may find two nodes with the same names and values. The first node of the data model is the document node that does not represent any thing in the document, but it refers to the document itself, then each node appears before its children which is called the document order. An example of XQuery data model:

```xml
<!- student information->
<student>
  <name age="27"/>
  <fname>ahmed</fname>
  <lname>abdel-aziz</lname>
  <academicYear>first year</academicYear>
</student>
```

This fragment represents a part of the document. The result of evaluating a path expression on a given XML document is considered as a tree where each node is the comment followed by the node element, name element, age attribute, fname element, text node containing "ahmed", the lname element, text node containing "abdel-aziz", address element, text node containing "neser city", academicYear element, and text node containing "first year".

2) Path Expression: In XQuery, path expressions [28] are used to select nodes from XML data. XQuery’s path expressions are derived from XPath (XML path Language) 1.0 and are identical to the path expressions of XPath 2.0. In XPath, an XML document is considered as a tree where each node represents a part of the document. The result of evaluating a Path expression on a given XML document is a set of nodes sorted according to the document order. A Path expression consists of one or more steps separated by slash (/) or double slash (/). A path expression may be absolute path that has the following form:

Path ::= /Step1/Step2/.../StepN

or relative path that has the following form

Path ::= Step1/Step2/.../StepN

The absolute path starts with a slash (/), and the current node is the root. The relative path does not start with a (/), but it starts with the current node which is the node where the expression is being used. A Path expression is evaluated sequentially from left to right, step by step. A Path step is applied to a single node (the current node) and selects a set of distinct nodes (based on node identity). Each node of the result node set is used as the current node to evaluate the following step. The result of evaluating a Path expression is the union of node sets selected by the last step. A path expression consists of path steps, the general form of the definition of a path step is shown as the following:

Step ::= Axis :: Nodetest [Predicate]*

Where the Axis specifies the direction in which the document should be searched, i.e. specifies the relationship between the nodes selected by the step and the current node in the XML tree. The following table shows twelve axes that are supported in the xpath language:
Examples of the axes are shown in the following table:

### TABLE III
**THE DIFFERENT TYPES OF AXES**

<table>
<thead>
<tr>
<th>Axis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>Includes all parent and grandparent of the current node. The root will not be included if it is the current node.</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>Includes the current node plus all its parent and grandparent.</td>
</tr>
<tr>
<td>descendant</td>
<td>Includes all children and grandchildren of the current node.</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>Includes the current node plus all its children and grandchildren.</td>
</tr>
<tr>
<td>parent</td>
<td>Includes the parent of the current node.</td>
</tr>
<tr>
<td>child</td>
<td>Includes all the children of the current node.</td>
</tr>
<tr>
<td>following</td>
<td>Includes everything in the document after the closing tag of the current node.</td>
</tr>
<tr>
<td>following-sibling</td>
<td>Includes all siblings after the current node.</td>
</tr>
<tr>
<td>preceding</td>
<td>Includes everything in the document that is before the starting tag of the current node.</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>Includes all siblings before the current node.</td>
</tr>
<tr>
<td>attribute</td>
<td>Includes all the attributes of the current node.</td>
</tr>
<tr>
<td>namespace</td>
<td>Includes all namespace nodes of the current node.</td>
</tr>
<tr>
<td>self</td>
<td>Includes the current node.</td>
</tr>
</tbody>
</table>

### TABLE IV
**EXAMPLES OF AXES**

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent::name</td>
<td>Selects the parent element that is called name of the current node. If the current node has no name parent node, it will select an empty node-set.</td>
</tr>
<tr>
<td>child::*</td>
<td>Selects all the child elements of the current node.</td>
</tr>
<tr>
<td>child::text()</td>
<td>Selects the text nodes that are children of the current node.</td>
</tr>
<tr>
<td>child::node()</td>
<td>Selects the children nodes (such as elements, comments, or PIs) of the current node.</td>
</tr>
<tr>
<td>child::/child::name</td>
<td>Selects all name grandchildren of the current node.</td>
</tr>
<tr>
<td>attribute::age</td>
<td>Selects the age attribute of the current node. If the current node has no age attribute, it will select an empty node-set.</td>
</tr>
<tr>
<td>attribute::*</td>
<td>Selects all attributes of the current node.</td>
</tr>
<tr>
<td>/</td>
<td>Selects the document root.</td>
</tr>
</tbody>
</table>

The Predicates are Boolean conditions that select a subset of the nodes from the node-set that computed by a step expression for which all predicates evaluate to true. A predicate consists of an expression called a predicate expression and enclosed in square brackets. We can have zero or more predicates in the steps of a path expression, for example:

**Example 1**: child::name[child::fname="abdel-aziz"]
This example selects all name elements that are children of the current node where fname element which is a child of the name element equals "abdel-aziz".

**Example 2**: child::name[position()=1]
This example selects the first child element of the current node that is called name.

**Example 3**: child::name[position()=last()]
This example selects the last child of the current node that is called name.

**Example 4**: child::name[position()=last()-1]
This example selects the child element of the current node that is called name that precedes the last child element.

**Example 5**: child::name[position()=<5]
This example selects the first four name children of the current node.

**Example 6**: /descendant::name[position()=5]
This example selects the fifth name element in the document.

**Example 7**: child::name[attribute::age="28"]
This example selects all children that are called name of the current node that have an age attribute with value "28". Abbreviations can be used when describing a location path. For example, the name of the axis which defaults to child can be omitted. If a predicate expression evaluates to an integer value, the value is considered to be the position of the node selected. Another abbreviation can be used is the empty step, //, that selects the current node and all of its descendant nodes.

The following table shows the abbreviations that can be used:

### TABLE V
**THE ABBREVIATION OF A PATH EXPRESSION**

<table>
<thead>
<tr>
<th>abbr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>child::</td>
</tr>
<tr>
<td>@</td>
<td>attribute::</td>
</tr>
<tr>
<td>.</td>
<td>self::node()</td>
</tr>
<tr>
<td>//</td>
<td>/parent::node()</td>
</tr>
</tbody>
</table>

In the following examples, we will explain the abbreviation of path expressions:

**Example 1**: child::name can be abbreviated to name which selects all the child elements that are called name of the current node.

**Example 2**: child::* can be abbreviated to * which selects all the child elements of the current node.

**Example 3**: child::name[attribute::age="27"] can be abbreviated to name[@age="27"] which selects all the child elements that are called name with attribute age equals "27" of the current node.

**Example 4**: attribute::* can be abbreviated @* which selects all the attributes of the current node.

**Example 5**: /descendant-or-self::node()/child::name can be abbreviated to /name which selects all descendants of the document root that are called name.

**Example 6**: self::node() abbreviated to . which selects the current node.

**Example 7**: self::node()/descendant-or-self::node()/child::name can be abbreviated to /name which selects all descendants that are called name of the current node.

**Example 8**: parent::node() can be abbreviated to .. which selects the parent of the current node.

**Example 9**: parent::node()/child::name can be abbreviated to ../name which selects the parent of the current node, then selects the name children of that parent.
Example 10: child::name[position()=4][attribute::age="28"] can be abbreviated to name[4]/@age="28" which selects the fourth child element that is called name if it has an age attribute equals "28" of the current node.

Example 11: child::name[attribute::age="27"][position()=4] can be abbreviated to name[@age="27"][4] which selects child element that is called name with an age attribute equals "27" and its position is the fourth child of the current node.

Example 12: /child::*//student[child::fname="abdel-aziz"]/child::address can be abbreviated to //student[fname="abdel-aziz"]/address which first selects all the children of the document root, second for each selected node for the first step, it selects all the child elements that are called student with a child fname equals "abdel-aziz", so the result until this step is a set of student elements where their first name is "abdel-aziz", and finally these student elements are used as the current node for the last step to select all the child address elements to the given student element.

3) FLWOR Expressions: Iteration is an important part of a query language. XQuery provides a way to iterate over a sequence of values, binding a variable to each of the values and evaluating an expression for each binding of the variable by using the FLWOR expression[2]. The FLWOR expression pronounced "flower" consists of:

- FOR-clause: binds one or more variables to a sequence of values returned by a path expression and iterates over the values. A for clause produces tuples of variable bindings that form the cartesian product of the iteration sequences.
- LET-clause: binds one or more variables to a sequence of values returned by path expression without iterating.
- WHERE-clause: is an optional clause that contains one or more predicates that select from the set-node that is generated by FOR-LET clauses, the set of nodes (tuples) that evaluate the predicates to "true".
- ORDER BY clause: to sort the binding tuples.
- RETURN-clause: generates the output of the FLWOR expression. The RETURN-clause usually contains one or more element constructors and/or references to variables. The return clause is executed once for each binding tuple that results from FOR-LET-WHERE-clauses, in order. The results of these executions are concatenated into a sequence as the result of the FLWOR expression.

The FLWOR expression is useful for computing joins between two or more documents and for restructuring data. The FLOWR expression is similar to SELECT-FROM-WHERE statement in SQL, but it is not defined in terms of tables, rows, and columns. Every clause in a FLWOR expression is defined in terms of tuples, and the for and let clauses create the tuples, therefore, every FLWOR expression must have at least one for or let clause. Now let us to have another XML document called bookForStudents.xml to give the examples:

Definition: A variable is a name preceded by $, and it can be bound to a scalar value or a sequence of nodes returned from a path expression.

Example 1: for $n in (3, 5)
return (tuple){$n} * 2/tuple
In this example, the for clause binds the variable ($n) to the expression (3, 5) which creates two tuples (the number of the iteration is the number of the values in the expression in the for clause) that are shown below:

( tuple ) 6 ( /tuple )
( tuple ) 10 ( /tuple )
The first tuple is according to binding the value "3" to ($n), and the second tuple is according to binding the value "5" to ($n). The order of the resulting tuple is the same as the order of the values in the expression (3, 5) in the for clause, so a for clause preserves order when it creates tuples. Note that the return clause contains an element constructor, so the variable ($n) must be enclosed in curly braces.

Example 2: for $m in (3, 5), $n in (2, 4)
return (tuple){$m} { $n} /tuple
In the above query, since the for clause binds two variables ($m) and ($n), we have two nested iterations where the outer iteration is for the variable ($m) and the inner iteration is for the variable ($n), so the result of the above query is four tuples that form the cartesian product of the two variable bindings ($m) and ($n), i.e. the result is:

( tuple ) (3, 2) ( /tuple )
( tuple ) (3, 4) ( /tuple )
( tuple ) (5, 2) ( /tuple )
( tuple ) (5, 4) ( /tuple )
The order of the resulting tuples is the same as the order of the variables binding, i.e. since each tuple is a combination of the variables ($m) and ($n), the order of the tuples based on the order of the values in the expression (3, 5) for the variable ($m), then for each value of ($m), the order based on the order of the values in the expression (2, 4) for the variable ($n).
Example 3: let $i$ in (1, 2, 3)
return (tuple) { $i$ } (/tuple)
In the above example, the variable ($i$) is bound to the values of the expression (1, 2, 3), and a single tuple is generated that contains the binding of the variable ($i$). This tuple is shown below:

(tuple)
(i) 1 (/i)
(j) 1 (/j)
(/tuple)

Example 4: for $i$ in (1 to 3)
let $j:=(1 to {i})$
return (tuple) (i) { $i$ } (/i)
(j) { $j$ } (/j)
(/tuple)
In the above example, the for clause binds the variable ($i$) to the values (1, 2, and 3) which is a sequence of integers that are generated by the to operator, and for each value binding to the variable ($i$), the let clause binds the variable ($j$) to a sequence of values from 1 to the value of the variable ($i$), and finally the return clause generates the result which is shown below:

(tuple)
(i) 1 (/i)
(j) 1 (/j)
(/tuple)

Example 5: for $s$ in //student
let $n:=($s/name$)$
where $n$="abdel-aziz"
return $s$/academicYear
In the above example, the for clause binds the result of the path expression to the variable($s$). In each binding, the variable ($s$) is bound to a (name) element, the where clause selects the binding tuples in which the (name) element of variable ($s$) equals "abdel-aziz", and finally the return clause creates the (student) element that contains the element (academicYear) of the variable ($s$).

Example 9: for $s$ in //name
order by $s$/lname descending , $s$/fname ascending
return $s$
In the above example, the for clause creates tuples where each one with an element (name). The order by clause sorts these tuples descending based on the value of the element (lname), and the return clause returns the (name) elements in the same order of the sorted tuples.
In the above example, The keyword collation means that the query sorts the student based on the ⟨lname⟩ element using a U.S. English comparison.

Example 11: for $a at $d in //distinct-values (author) return (authorList)
number {($d)} is (author){($a)}/author)
/authorList)
In the above example, the variable ($d) is called a positional variable. The name of the positional variable is preceded by the keyword at. The variable ($d) iterates over the number of the element (author) in the document starting with 1. The value of the variable ($d) represents the position of the element (author) in the document. The distinct-values() function extracts the values of a sequence of author nodes and creates a sequence of unique author names by eliminating duplicates. The result of the above query is

Example 12: for $s in doc("bookForStudent.xml")/book
for $b in doc("student.xml")/student
where $s/academicYear=$b/academicYear
return (student)
{($s)/name
$gender=$s/academicYear
$b/title}
In the above example, first we use the doc() function which returns the document node for the document named "student.xml" and the document node for the document named "bookForStudent.xml", then in the for clause we bind the document node of "student.xml" to the variable ($s) and bind the document node of "bookForStudent.xml" to the variable ($b), second the for clause creates tuples from the cartesian product of the two variable ($s) and ($b), the where clause selects the tuples where the ⟨academicYear⟩ element of the variable ($s) matches the ⟨academicYear⟩ element of the the variable ($b), and finally the return clause creates a (student) element contains the (name) element, the (academicYear) element, and the (title) element of the book.

B. XQuery and Relations

The selection and projection of XQuery and SQL may seem very similar from the outside, but as described in [12], dealing with the structure of XML combined with flat relational tables is something that must be alluded. Another distinction between relational tables and XML, that must be pointed out, is the inherent relationship between nodes in a document and the necessity to express relationships between tables in form of SQL views. This also affects the relational XML encoding in a manner that not all relationships are directly present as attributes and have to be reconstructed via over relational joins. Bringing up the Data Model features of XML again, relational database tables are considered to be like sets in a mathematical manner having no direct ordering other than those possibly derived from their values. XML data though has an intrinsic order and each node has a unique node identity. This order is not derived from its data values. Instead XQuery must retain the original ordering based on the source data. This also accounts for sequences (a0,a1...an) which would be returned in a query:

FOR $x in (1,2,3) RETURN $x
in the exact order, namely 1,2,3. The concept is known as sequence order in XQuery.

On the other hand, consider the following table:

<table>
<thead>
<tr>
<th>rank</th>
<th>country</th>
<th>gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>USA</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>China</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>Russia</td>
<td>27</td>
</tr>
</tbody>
</table>

an SQL query like:

SELECT rank, country, gold
FROM Athens2004
WHERE gold >20
would not necessarily give back the three tuples ordered by its ranking, even if the tuples explain their ordering by themselves, but they must be explicitly defined by an ORDER BY rank clause. The problem also appears when nested FLWOR expressions occur and in this case the ordering becomes very important. Consider the following query:

FOR $x in (1,2)
RETURN
FOR $y in (10,20) RETURN $x+$y

The result of this query is easy to compute by hand, taking the iteration of the first value bound to $x of the outer sequence, iterating through the inner sequence and adding each bound value of $y to the first bound value of $x, receiving the values 11, 21, then returning to the outer iteration and performing the same procedure for the next value bound to $x and so on until we receive all four result values 11, 21, 12, 22. The problem here is how to perform such a trivial query in a relational form. How can we teach a relational system to perform such iterations and which values are to be added to each other? An idea that comes to mind is to remember each iteration
for every sequence of items and even every single item as a sequence of length one.

Looking again at the query, obviously two iterations must be stored for the outer and inner loops, in this case possibly numbered as 0.1 and 1.2. The idea here is related to the terms of loop lifting from[12]. If, for instance, a sequence like (10,20) in Query 2.1 is in an inner loop, then it must expose all iterations from the outer (0,1) loop together with its own iterations (1,2).

The nested loops are mapped into a small relation in Table 6. The left column marks the outer iteration, the middle column the inner. At the end, a sequence of items should be present looking like in Table 7, where the outer loop values, all set to 0, express a flattened interpretation of the nested query result. This example is very simple, but XQuery allows arbitrary nesting of expressions which makes it difficult to see how such nesting could be compiled without introducing scoping. In functional languages, variables with bound values can appear elsewhere in so called scopes. Such scopes are opened in FOR clauses setting a scope, but they also are opened by LET. We give a general notation for a FOR clause opened in FOR clauses setting a scope, but they also are

An occurrence of $x$ in expression e would mean that $S_0$ is its valid scope. The same applies to $y$ in the second FLWOR expression in the sequence, where $S_1$ is identified as its valid scope. In $S_1$ though, only $y$ is in a valid scope and the occurrence of $x$ would yield back a scoping error. The scopes $S_0$ and $S_1$ are not nested, but separated by the top level sequence expression, so neither of them have any scoped variables in common. Again in $S_0$, if a further FLWOR expression occurred, a child scope marked $S_{0.1}$ would be opened as a nested scope of $S_0$. The notation $S_{x,y}$ with $x \in \{0,1\}^\ast$ and $y \in \{0,1\}$ identifies the child scope $y$ of parent scope $x$.

C. XQuery to SQL Translation

Now, we will present the survey about the existing XQuery-to-SQL algorithms that depend on special XML-to-RDB schema-mapping algorithm and that doesn’t.

1) XQuery-to-SQL Translating Algorithm with Little Dependence on Schema Mapping between XML and RDB: in [13], the authors have been proposed an XQuery-to-SQL translating algorithm with little dependence on any schema-mapping algorithm between XML and RDB. The main ideas are as follows. Firstly, setting up a General Virtual Relational Schema for XML Document (GVRS for short) model as the bridge between XML instance document and factual RDB. The GVRS can describe any XML instance document from the point of E-R. And then the XQuery-to-SQL translating algorithm is achieved in two steps: translating XQuery on XML View to Virtual SQL (VSQL for short) on GVRS, then translating VSQL on GVRS to SQL on the factual bottom RDB. The algorithm of setting up GVRS model from the XML instance document is shown in the following Figure:

For example, consider the following XML document:

```
<article>
  ⟨author ID= 1⟩
    ⟨fname⟩Ahmed ⟨/fname⟩
    ⟨lname⟩Abdelaizaiz ⟨/lname⟩
    ⟨address⟩
      ⟨city⟩Giza ⟨/city⟩
      ⟨Zip⟩11112 ⟨/Zip⟩
      ⟨/address⟩
  ⟨/author⟩
  ⟨title⟩Selfish Gene ⟨/title⟩
  ⟨address⟩
    ⟨city⟩Giza ⟨/city⟩
    ⟨Zip⟩11112 ⟨/Zip⟩
  ⟨/address⟩
</article>
```

the corresponding GVRS model is:

```
element(eID, pID, Tag, Type, Value, T, C )
attribute(aID, eID, Tag, Type, Value, T, C )
```

In the second step, XQuery is translated on XML documents to VSQL on the GVRS and then VSQL is translated on GVRS to SQL on the factual relational schema. The algorithm from Path Expression to SQL is given below:

```
TRANSLATEXMLtoGVRS(XMLDocument)
BEGIN: pID=O;
IF CurrElen has attribute node THEN INSERT into Attribute;
IF CurrElen has value node THEN modify R.Vulue and R.Type;
IF CurrElen has sibling node THEN BEGIN
  childNode = the first child node of CurrElen;
  pID = childNode
END
WHILE childNode is an element node DO BEGIN
  childNode = the next sibling node of childNode
END
WHILE childNode is an element node DO BEGIN
  TranslateXMLtoGVRS(childNode)
```

Fig. 10. XML document

Fig. 11. GVRS model
The purpose is mostly to clean up the IR generated by the translator. Adjacent FROM, WHERE, and ORDER BY clauses are consolidated, making it easier for the SQL generator to navigate the IR tree.

**Phase II : SQL Generation**

Here the optimizer chooses how to partition the IR tree in order to generate SQL queries, decides what computations to do in the middleware, and optimizes loop invariants. Partitioning The purpose of this step is to partition the IR tree into connected components, by removing some of the xmltag nodes.

**Phase III : SQL Rewriting**

This phase is the most extensive one consisting of several rewrite rules designed to optimize the often inefficient generated SQL queries. The heuristic is to apply a rewriting if it decreases the cost of the query according to the optimizer 3. If the necessary hook is not provided to access the optimizer, we attempt to make the query as flat as possible. Generally, only limited capability engines can not return costs and so this heuristic is usually an improvement.

3) Isolating order semantics in order-sensitive Xquery-to-SQL translation: In [24], Order is essential for XML query processing. Efficient XML processing with order consideration over relational storage is non-trivial, especially for complex nested XQuery expressions. The order semantics may impede efficient query rewriting for nested query blocks. The authors have proposed a general order-sensitive XQuery processing approach involving three steps. First an algorithm is proposed for inferencing about and then isolating the order semantics in XQuery expressions specified over virtual XML views. This turns an ordered XQuery plan into an unordered one decorated with minimized order context annotations. Then without loss of semantics, logical optimization via XQuery rewriting can be easily applied to this transformed query plan. As last step, the translation of the optimized logical plan into SQL now correctly incorporates the order context annotations to assure the original order semantics.

IV. CONCLUSION

In this paper, we present a survey about the recent and previous mapping approaches that map XML documents to RDB. This survey will support the future research and development work as well as to raise the awareness for the presented approaches. In this survey, we study how the various definitions in a given XML DTD, such as elements, attributes, parent-child relationships, and ID-IDREF(s) attributes can be mapped to entities and relationships. How algorithms which map DTD to relational schema and as well as content and structure preserve the functional dependencies during the mapping process in order to produce relations with less redundancy. Finally, How the recent and previous approaches translate XML query languages, such as XQuery and xpath languages into SQL.
REFERENCES