International Journal of Cooperative Information Systems
Vol. 24, No. 4 (2015) 1550005 (29 pages)

DOI: 10.1142/S0218843015500057

World Scientific
www.worldscientific.com

Designing Redundancy-Free XML Schema: A Smallest Closure Approach

Wai Yan Mok*,‡, Joseph Fong†,§ and Kenneth Wong†

*College of Business Administration
University of Alabama in Huntsville
Huntsville, AL 35899, USA

†Computer Science Department
City University of Hong Kong
Hong Kong, P. R. China
‡mokw@uah.edu
§csjfong@cityu.edu.hk

Received 7 February 2013
Accepted 28 August 2015
Published 12 October 2015

XML has too many low-level details that hinder high-level conceptual design. We therefore propose DTD graphs and XSD graphs as a mean for conceptual modeling of XML applications. Similar to document type definitions (DTDs) and XML schema definitions (XSDs), DTD graphs and XSD graphs are trees, and as such they can easily be mapped to DTDs and XSDs for implementation. Unlike DTDs and XSDs, DTD graphs and XSD graphs capture various high-level data semantics such as cardinality, ISA, participation, aggregation, categorization, and n-ary relationship. Furthermore, this paper also presents transformation rules between DTDs and a large class of XML Schemas and an algorithm that inputs users' requirements and outputs a DTD graph that has a minimum number of redundancy-free fragments. As a result of these good properties, the resulting DTD or XSD facilitates query processing and update.

Keywords: DTD graphs; XSD graphs; DTDs; XSDs.

1. Introduction

XML data have been freely exchanged over the Internet nowadays. Many exchange standards exist for different applications. A simple search on the phrase "XML markup languages" on the Internet will show page after page of XML languages defined for various industries. For example, eXtensible Business Reporting Language (XBRL)[a] is a global standard for exchanging business information. One major use of XBRL is to define and exchange financial statements, which require strict exchange standards. Another example is eXtensible Access Control Markup Language (XACML).[b] One of the stated goals of XACML is to promote common

[a]https://www.xbrl.org/.
[b]http://xml.coverpages.org/xacml.html.

terminology and interoperability between access control implementations by multiple vendors. Yet another example is the HL7 Clinical Document Architecture (CDA),[c] which is a XML-based markup standard intended to specify the encoding, structure and semantics of clinical documents for exchange. Because of the flexibility of XML, we expect more XML markup languages will be developed and this trend will continue.

An important first step of the development of any large-scale software requires understanding and documenting the application at hand. The development of all the aforementioned XML markup languages is no exception. We believe that conceptual modeling is able to help develop XML markup languages. As an observation, the long research history of relational databases demonstrates that conceptual modeling of relational databases is an essential step of the development process. Along the same line of reasoning, an XML conceptual model that facilitates conceptual modeling of XML applications has become a necessity. It is our belief that DTD graphs and XSD graphs are able to fill this need. This paper first presents an overview of the high-level data semantics offered by DTD graphs and XSD graphs. It then provides transformation rules between DTDs and a large class of XML Schemas. This class of XML Schemas is important because, like DTDs, it captures a large collection of real-world cases. After which, we present an algorithm that inputs users' requirements and outputs a DTD graph that has a minimum number of redundancy-free fragments, and prove its correctness. Because the generated DTD graph has the minimum number of redundancy-free fragments, the number of reference pointers is also minimum, which means that extracting data across the fragments can be quickly done.

Figure 1 shows an overview of the proposed XML application design process of this paper. Users' requirements of an XML application are first provided as part of the conceptual modeling step. The result of which is a DTD graph or an XSD graph, depending upon whether a DTD or an XSD is the desired output. While the semantics of the application are captured in the DTD graph or the XSD graph, the

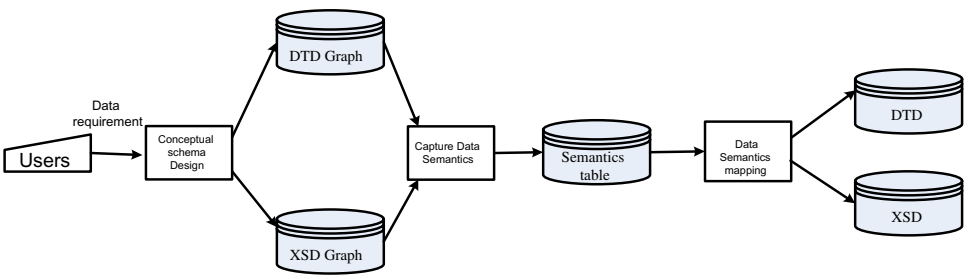


Fig. 1. An overview of the proposed XML design process.

[c]http://www.hl7.org/index.cfm?ref=nav.

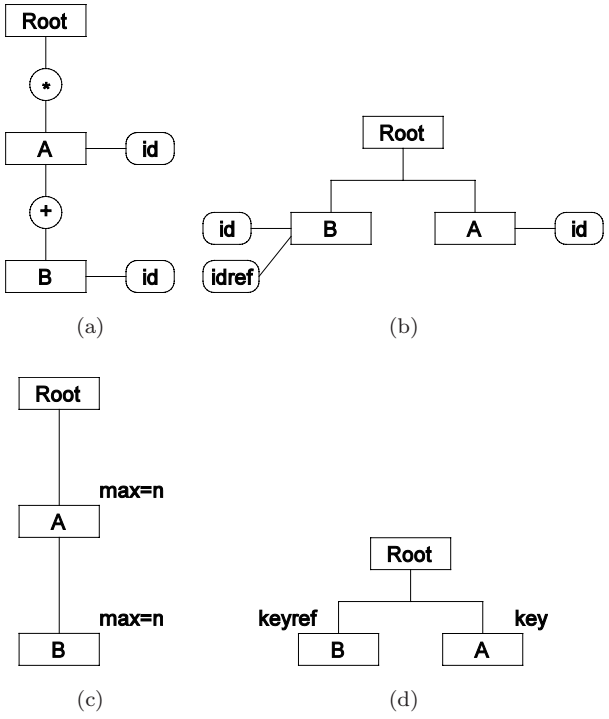


Fig. 2. Four different XML database designs.

semantics table is an essential data structure that expedites code generation. The final product is a DTD or an XSD ready to verify XML documents.

As a motivating example for this paper, consider Fig. 2 that shows four different XML database designs. There are two DTD graphs shown in Figs. 2(a) and 2(b); one of which has a reference pointer and the other does not. Figures 2(c) and 2(d) show two similar XSD graphs; one of which has a reference pointer and the other has not. If the relationship between A elements and B elements is many-to-one, then all B elements that are associated with an A element can be clustered with the A element with no redundancy. The implication of which is that retrieving the B elements of an A element can be quickly done because there is no need to traverse reference pointers. Thus, Figs. 2(a) and 2(c) illustrate good designs. On the other hand, if the relationship between A elements and B elements is many-to-many, a B element may associate with more than one A element and thus the designs shown in Figs. 2(a) and 2(c) will have redundancy. In this case, Figs. 2(b) and 2(d) show designs that avoid redundancy, although traversing reference pointers to retrieve the B elements for a particular A element has become necessary. (As a minor note of technicality, in the case that the relationship between A elements and B elements is many-to-many, idref of B in Fig. 2(b) should be read as idrefs to emphasize that a B element can associate with more than one A element.)

This paper is different from our previous work[10,11] in terms of the inputs and the outputs of the algorithms. While the inputs of Refs. 10 and 11 are respectively acyclic hypergraphs and conceptual-model hypergraphs, the inputs for this paper are a set of object types and a set of data semantics over the object types. The outputs of this paper are DTD graphs. On the other hand, the outputs of Refs. 10 and 11 are scheme trees.

The remainder of the paper is organized as follows. Section 2 introduces DTD graphs and XSD graphs. Section 3 provides a set of transformation rules between DTD graphs and a subclass of XSD graphs. For this subclass of XSD graphs, the set of transformation rules in Sec. 3 can easily transform a graph of one type to another. Section 4 presents the main algorithm of the paper, which outputs a DTD graph that will not lead to redundant data and has a minimum number of fragments from user's requirements. The correctness of the algorithm will also be proved there. An extensive example will be presented in Sec. 5, experiments in Sec. 6, and conclusions in Sec. 7.

2. Related Work

2.1. *DTD graphs*

Funderburk *et al.* introduced DTD graphs in 2002.[5,6] In essence, a DTD graph is a graphical representation of a DTD. Notationally, rectangles in DTD graphs represent XML elements and circles represent XML attributes. DTD graphs also employ the usual cardinality operators: "?", "∗", "+", and "|". A DTD graph is organized as a tree, in which parent–child relationships are shown explicitly. An example of DTD graphs now follows.

Figure 3 shows a sample DTD graph. "Patient Record" is the root element of the DTD graph, and it has a single child element "Patient." A "Patient" element has 0 or more "Record Folder" child elements. In turn, a "Record Folder" element has 0 or more "Medical Record" child elements. A "Medical Record" is either an "AE Record," or a "Ward Record," or an "Outpatient Record."

2.2. *XSD graphs*

Fong introduced XSD graphs in Ref. 3, and the formation rules of which can be found in Refs. 3 and 4. In essence, XSD graphs are able to visualize, specify, and document structural constraints of XSDs. The resulting graph can be used to represent the relationships of the elements in an XSD, together with various data semantic specifications. A brief summary of XSD graphs is provided here, where the details can be found in Refs. 3 and 4.

Figure 4 shows the XSD graph constructs, each of which is discussed as follows:

(a) E_r is an XML element that denotes a binary many-to-many relationship between two other XML elements.

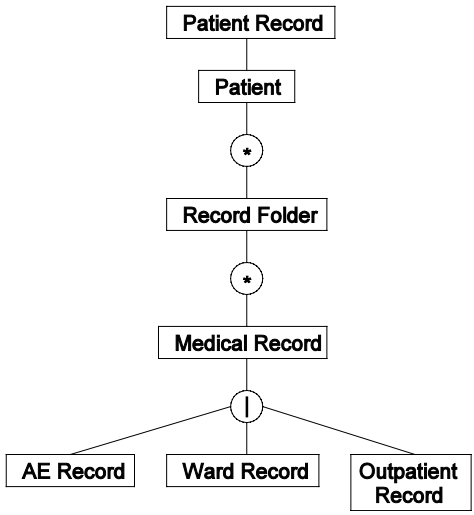


Fig. 3. A patient record DTD graph.

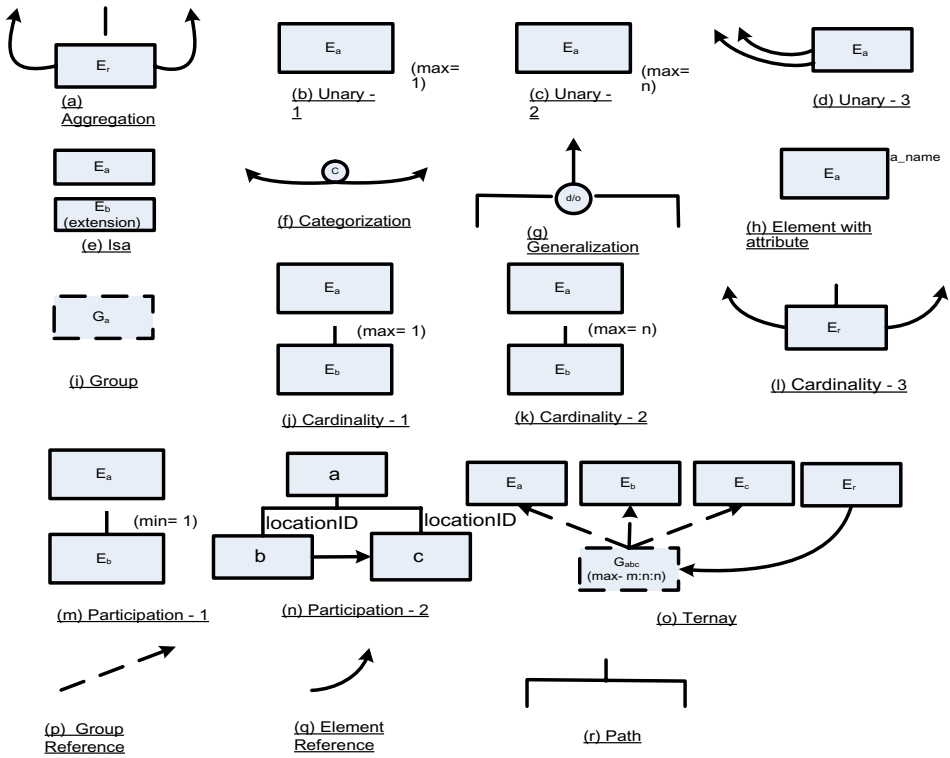


Fig. 4. XSD graph constructs.

(b) E_a is an XML element that denotes a unary one-to-one relationship from E_a to itself.
(c) E_a is an XML element that denotes a unary one-to-many relationship from E_a to itself.
(d) E_a is an XML element that denotes a unary many-to-many relationship from E_a to itself.
(e) E_a and E_b are XML elements that denote an extension relationship, or an is-a relationship, in which E_b inherits all properties of E_a.
(f) An XML element tagged with the keyword "choice" is in an is-a relationship with every XML element pointed at by an arrow emanated from the keyword "choice."
(g) "d" or "o" denote disjoint or overlap generalization.
(h) E_a represents an XML element with an attribute declaration.
(i) G_a represents a group declaration.
(j) E_b is a child XML element that is in a one-to-one relationship with its parent XML element E_a.
(k) E_b is a child XML element that is in a many-to-one relationship with its parent XML element E_a.
(l) E_r is an XML element that refers to two other XML elements that are in a many-to-many relationship.
(m) With "min = 1", XML element E_b is in a total participation with its parent E_a.
(n) Without "min = 1", XML element E_b is in a partial participation with its parent E_a.
(o) E_r is an XML element that denotes a ternary relationship among the XML elements E_a, E_b, and E_c.
(p) A broken-line arrow represents a "ref" keyword for a group declaration.
(q) A concrete-line arrow represents a "ref" keyword for an element declaration.
(r) A hierarchy path shows one parent XML element with two child XML elements.

2.3. *XML data normalization*

Much work has been done in XML normalization.[1,7,9,12,13,15,16] Mok and Embley[9] proved that generating the fewest redundancy-free XML scheme trees from hypergraphs is NP-hard. However, if the universal-relation-scheme assumption[8] holds for a hypergraph H, and if H is Graham-reduction acyclic,[8] and if each hyperedge in H is in BCNF,[8] then Mok *et al.*[10] proved that extracting the largest redundancy-free XML scheme tree from H can be done in polynomial time.

According to the methodology in Ref. 13, functional dependencies and normal forms for XML can be used to create web-based systems on the Internet. Vincent *et al.*[13] also proposed a redundancy-free normal form, namely XNF, for XML. They, however, did not consider data semantics and the whole picture of tree view in their XML Trees.

Fong *et al.*[4] proposed XSD graphs (derived from XML Tree Model) with data semantics of cardinality, participation, is-a, generalization, categorization, aggregation, u-ary and n-ary relationship as XML conceptual schema and provided algorithms that map XSD Graphs to XSDs for implementation.

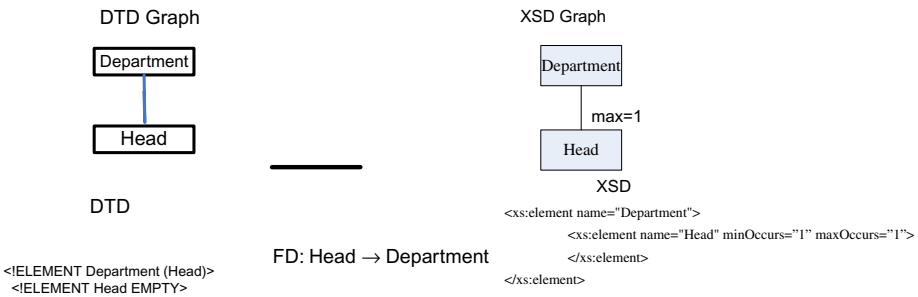
2.4. *Resource description framework*

The RDF data model is similar to the classic conceptual modeling approaches such as entity–relationship or UML class diagrams, as it is based upon the idea of making statements about resources in the form of subject–predicate–object expressions. DTD graphs, however, focus on communications with clients and thus graphical notations seem to be a better choice than subject–predicate–object expressions. While subject–predicate–object expressions are easier for machines to parse and understand, they are hard for humans. We believe DTD graphs and XSD graphs are easier for humans to read and understand.

3. Transformations Between DTD Graphs and a Large Class of XSD Graphs

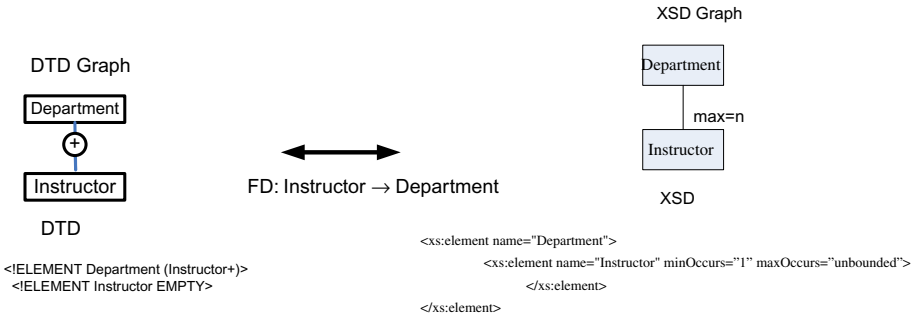
It is well known that XSDs are more powerful than DTDs because among many things, XML Schemas support data types and namespaces that cannot be done in DTDs.[14] However, this section defines a class of XML Schemas, which are equivalent to DTDs in the sense that each DTD can be transformed to an XSD in this class; and similarly every XSD in this class can be transformed to a DTD. Like DTDs, this class of XML Schemas is interesting because it captures a large class of real-world cases. Most importantly, the smallest closure approach proposed in this paper applies immediately to this class of XML Schemas. Hence, this paper focuses on this class of XML Schemas and DTDs.

Figure 5 presents a transformation rule for each basic DTD graph construct and its corresponding XSD graph construct. The transformation rules are rather straightforward and thus the proofs of which are omitted. Instead, intuitive

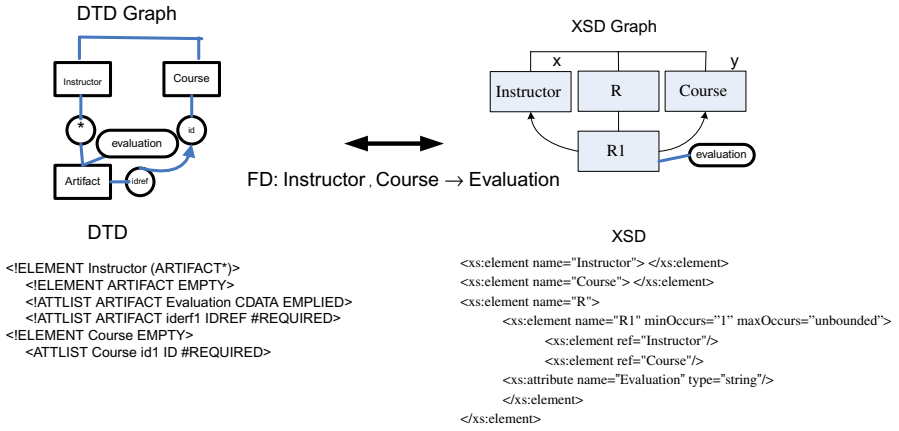


(a) One-to-one cardinality between Element Department and Element Head.

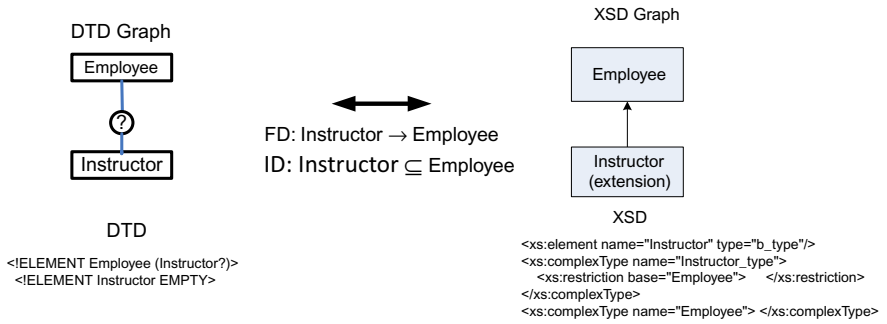
Fig. 5. Transformations between DTDs and a large class of XML schemas.



(b) One-to-Many cardinality between Element Department and Element Instructor.

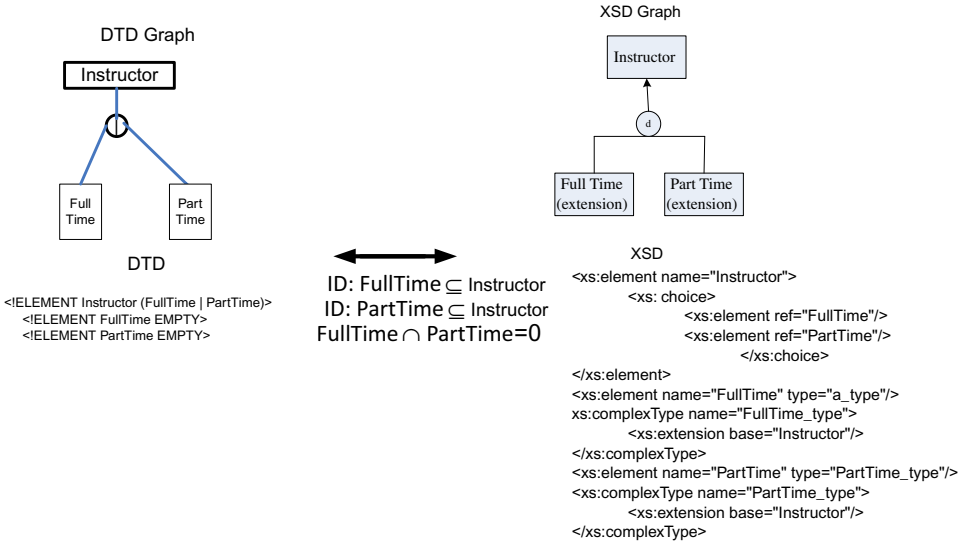


(c) Many-to-many cardinality between Element A and Element B with attribute c.

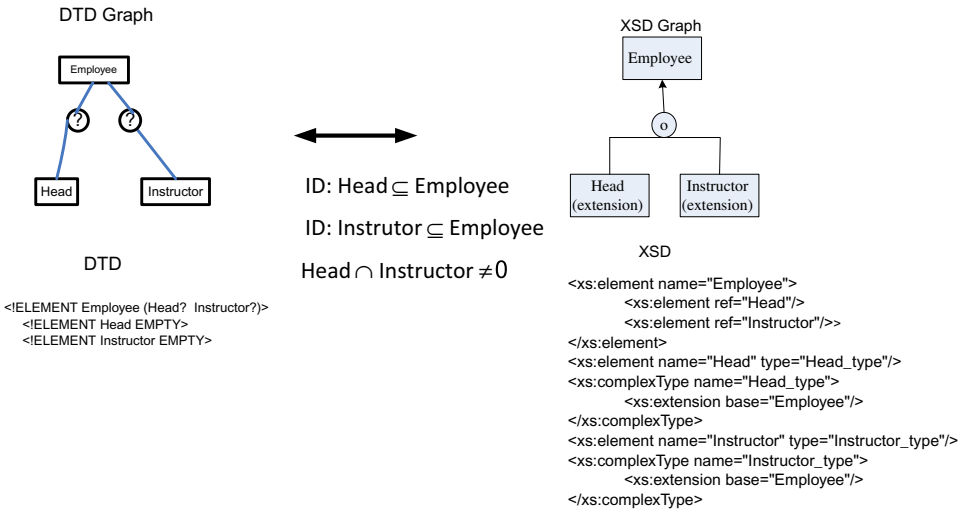


(d) Is-a relationship between Element Employee and Element Instructor.

Fig. 5. (*Continued*)

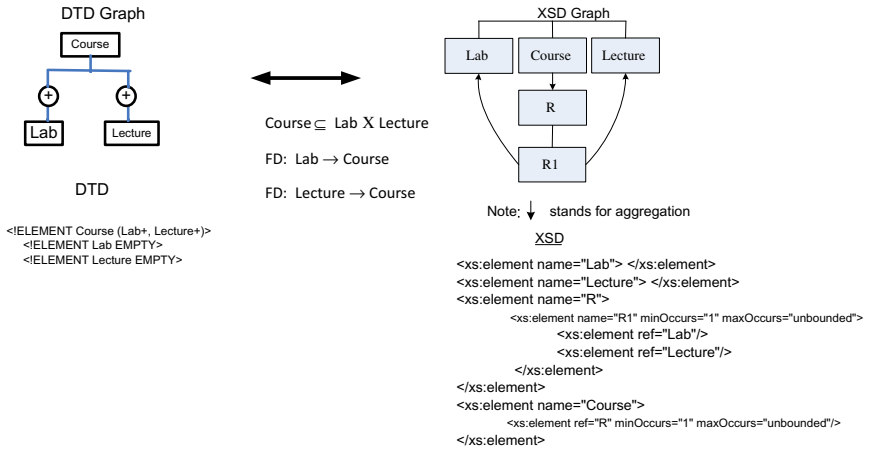


(e) Disjoint Generalization of subclass Elements FullTime and PartTime under superclass Element Instructor.

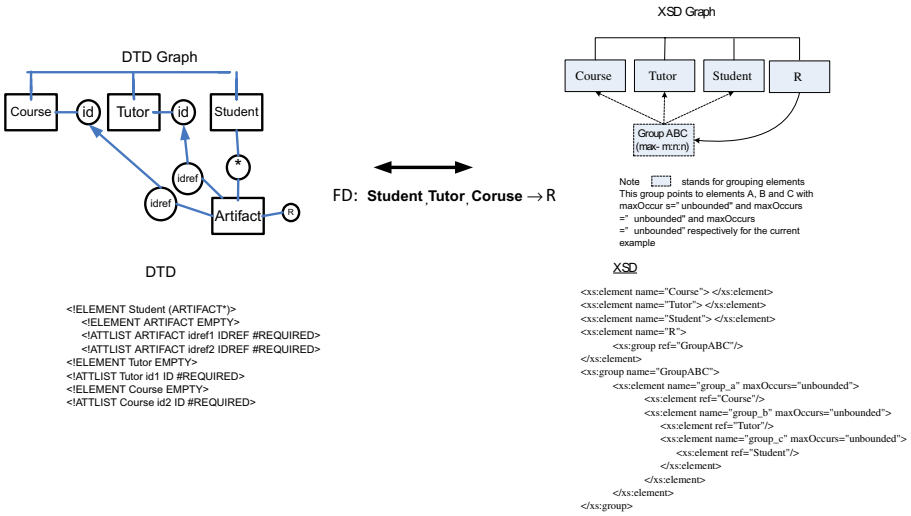


(f) Overlap Generalization of subclass Elements Head and Instructor under superclass Element Employee.

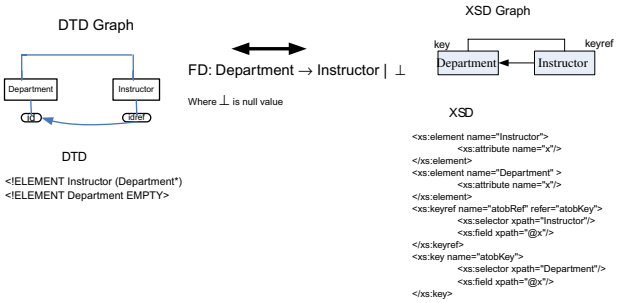
Fig. 5. (*Continued*)



(g) Aggregation of Element Course with component Elements Lab and Lecture.

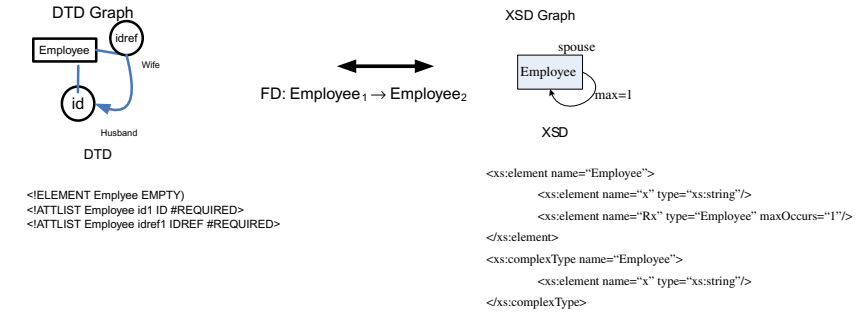


(h) N-any relationship among Elements Course, Tutor and Student with attribute R.

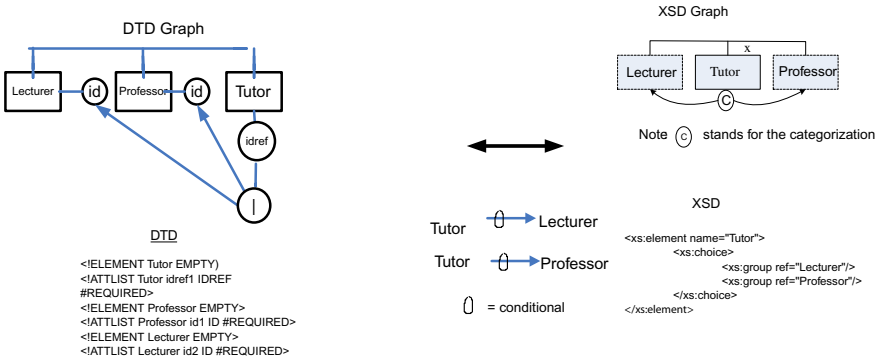


(i) Partial Participation between Element Department and Element Instructor with null value ⊥.

Fig. 5. (*Continued*)



(j) u-ary relationship for element Employee.



(k) Categorization of Elements Lecturer and Professor to Element Tutor.

Fig. 5. (*Continued*)

examples are provided instead to demonstrate the transformations. Since there is a one-to-one correspondence between this class of XML Schemas and DTDs, from this point on this paper focuses on DTD graphs. Note that the DTD graph constructs in Fig. 5 will be further used in the proof of correctness of the main algorithm of the paper.

4. An XML Schema Generation Algorithm

This section first presents the definitions of basic concepts and assumptions. After which, the main algorithm of the paper and the proof of its correctness will be given.

Definition 1. A fragment in a DTD graph is a child element of the root element of the DTD graph.

For this paper, all real-world object types have a corresponding XML element definition. Let O be an object type and E be its corresponding XML element definition. Attributes of O are the only attributes of E. Each instance of O is represented by at least one instance of E in an XML document. In addition, each object type O

must have a special attribute, called Oid, which is an identifier for O. Each instance of O must have a unique value over Oid.

Definition 2. A data value v of an XML document D is redundant if there is no loss of information in D if v was removed from D.

In terms of functional dependencies (FDs), the following definition provides the details on how redundant data values were caused by FDs.

Definition 3. An attribute value v over an attribute A in an XML document D is redundant with respect to an FD $X \rightarrow A$ if there are two distinct XML element instances I_1 and I_2 in D such that $I_1(XA) = I_2(XA)$. Then, either one of the two values $I_1(A)$ or $I_2(A)$, but not both, is redundant.

XML element instances and object instances are two different concepts and must not be misunderstood that they are equivalent. Regardless of its contents, every XML element instance is unique because it is identified by a unique XPath expression pointing at its location in a XML document. Although two or more XML element instances may represent the same object instance, they are all distinct.

Example 1. For the XML document in Fig. 6, student John with the Oid s01 is a member of both chess club and drama club. Therefore, his data are stored in two different XML element instances. Note that his Oid is stored in both STUDENT element instances. To emphasize that the two STUDENT element instances are different, we add an attribute Eid of type ID to the STUDENT element definition.

```
<?xml version="1.0"?>
<!DOCTYPE CLUB [
<!ELEMENT CLUB (CHESSCLUB, DRAMACLUB)>
<!ELEMENT CHESSCLUB (STUDENT*)>
<!ELEMENT DRAMACLUB (STUDENT*)>
<!ELEMENT STUDENT (#PCDATA)>
<!ATTLIST STUDENT EID ID #REQUIRED>
<!ATTLIST STUDENT OID CDATA #REQUIRED>
<!ATTLIST STUDENT ADDRESS CDATA #REQUIRED>
]>
<CLUB>
  <CHESSCLUB>
    <STUDENT EID="e01" OID="s01" ADDRESS="201 Oak st.">
      John
    </STUDENT>
  </CHESSCLUB>
  <DRAMACLUB>
    <STUDENT EID="e02" OID="s01" ADDRESS="201 Oak st.">
      John
    </STUDENT>
  </DRAMACLUB>
</CLUB>
```

Fig. 6. An XML document with data redundancy.

Each STUDENT element instance must have a unique value for Eid; otherwise, the XML document is invalid.

Note that FDs naturally arise because of object identifiers. Let O be an object type and Oid, $A_1, A_2, \ldots, A_n$ be its attributes where Oid is the object identifier of O. By the nature of object identifiers, every object type O gives rise to an FD Oid $\rightarrow A_1A_2 \ldots A_n$.

Example 2. There are redundant attribute values in the XML document in Fig. 6 caused by the FD Oid $\rightarrow$ Address. Since both STUDENT element instances have the same Oid value, they represent the same object instance and thus the student's address is stored twice. Hence, one of the two ADDRESS values is redundant with respect to the FD Oid $\rightarrow$ Address.

Lemma 1. *Let O be an object type and E be its corresponding XML element definition. Let* Oid, $A_1, A_2, \ldots, A_n$ *be the attributes of O. Let D be an XML document. D has redundant data values with respect to* Oid $\rightarrow A_i$ *if and only if there are two distinct instances* I_1 *and* I_2 *of E that represent the same instance o of O and they have at least two common attributes and one of which is* Oid.

Proof. Suppose D has redundant data values with respect to Oid $\rightarrow A_i$. By Definition 3, there are two element instances I_1 and I_2 in D such that both I_1 and I_2 have the same values over the attributes Oid and A_i. Since I_1 and I_2 have the same value of Oid, they represent the same instance o of object type O. Since E is O's corresponding XML element definition, I_1 and I_2 are instances of E. Since $I_1(\text{Oid}) = I_2(\text{Oid}) = o$, thus $I_1(A_i) = I_2(A_i)$ by the FD Oid $\rightarrow A_i$. This means that one of the values $I_1(A_i)$ and $I_2(A_i)$, but not both, is redundant. On the other hand, if there are two distinct instances I_1 and I_2 of E that represent the same instance o of O and they have at least two common attributes and one of which is Oid, then I_1 and I_2 satisfy the FD Oid $\rightarrow A_i$ where A_i is the other attribute that I_1 and I_2 have in common. By Definition 3 again, D has redundant attribute value with respect to Oid $\rightarrow A_i$. □

In this paper, we assume the set S of attributes of every object type, or its corresponding XML element definition, is in BCNF (Boyce–Codd normal form).[8] This assumption is reasonable because most object types have very few attributes. If, however, an object type O is not in BCNF, it can always be decomposed into multiple BCNF object types. Hence, we can safely make this assumption.

Example 3. There are two FDs and four attributes in Fig. 7. The left-hand sides of the FDs are Sid and Address Name, which are both keys for the attributes of the XML element definition STUDENT. Hence, the XML element definition STUDENT is in BCNF and thus the DTD graph of Fig. 7 is in BCNF.

Example 4. There is an FD Home $\rightarrow$ HomePhone in Fig. 8(a) over the attributes of the XML element definition STUDENT. Since Home is not a key, the DTD

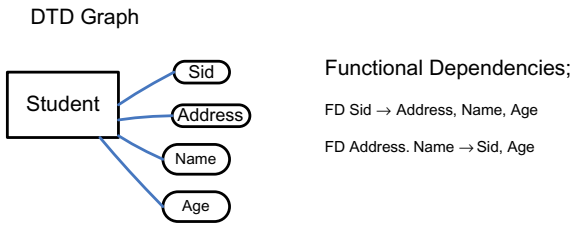
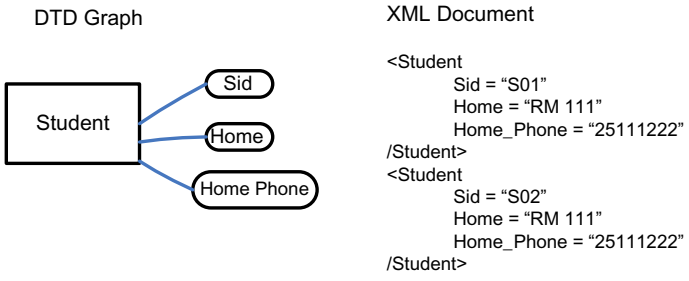
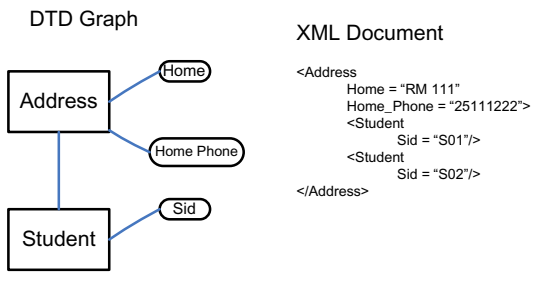


Fig. 7. A BCNF DTD.



(a)



(b)

Fig. 8. (a) Non-BCNF DTD and (b) BCNF DTD.

graph in Fig. 8(a) is not in BCNF. (It is possible that more than one student resides in the same home.) However, the XML element definition STUDENT can be easily decomposed into two BCNF XML element definitions, which are shown in Fig. 8(b). Now, the FD Home → HomePhone only applies to the XML element definition ADDRESS, and Home is a key of the attributes of ADDRESS. Thus, both XML element definitions ADDRESS and STUDENT are in BCNF.

Lemma 2. *Let O be a BCNF object type and E be its corresponding XML element definition. Let* $\mathrm{Oid}, \mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_n$ *be the attributes of O. Let D be an XML document. D has redundant data values with respect to a non-trivial FD* $\mathrm{X} \rightarrow \mathrm{A}_i$ *where* $\mathrm{XA}_i \subseteq \mathrm{OidA}_1\mathrm{A}_2 \ldots \mathrm{A}_n$ *if and only if D has redundant data values with respect to* $\mathrm{Oid} \rightarrow \mathrm{A}_i$.

Proof. Since O is in BCNF and $X \rightarrow A_i$ is non-trivial, X is another key of O. Hence, $X \rightarrow Oid$. Thus, the proof of this lemma is similar to that for Lemma 1. □

We now present Algorithm 1, the main algorithm of the paper.

Algorithm 1. An algorithm that outputs a DTD graph using the smallest closure approach.

Input: A set S of XML elements that represent a collection of real-world BCNF object types, and a set of data semantics over the elements in S.
Output: A DTD graph G (G has only the root element before Algorithm 1 is executed.).

Step 1. Extract FDs from the input, but discard all FDs of the form X o $\rightarrow$ Y where the symbol o $\rightarrow$ denotes that some instances of X might not participate in the FD with the instances of Y; i.e. there might be some X instances that do not have associated Y instances.[d]

Step 2. Generate functional closures for all XML elements in S using the FDs extracted in Step 1.

Step 3. Repeat until all XML elements in S have been added to G.

Move each remaining smallest XML element in S to G. (An XML element A is smallest if there is no other XML element B in S such that $B^+ \subset A^+$.) However, if there are smallest XML elements $A_1, A_2, \ldots, A_n$ such that $A_1^+ = A_2^+ = \cdots = A_n^+$, then move all of them $A_1, A_2, \ldots, A_n$ to G. After which, make A_1 a child element of an XML element which was added in the last iteration.

Add the input data semantics of the XML elements to G.

Step 4. For every XML element A that is a child of more than one XML element in G, replace all but one occurrence of A by another XML element Aref that only contains a reference pointer idref that refers to the id of A; i.e. Aref simply acts as a pointer pointing to A in G.

Theorem 1. *Algorithm* 1 *generates a redundancy-free DTD graph from the input semantic metadata.*

Proof. Let H be the input semantic metadata and G be the output DTD graph. We need to show that for any XML document D of G, D is free of redundant attribute values. Let O be a BCNF object type and E be its corresponding XML

[d]The symbol X o $\rightarrow$ Y means that the instances of X optionally associate with the instances of Y in the FD. In other words, there might be instances of X that do not associate with any instances of Y in the FD. The example in Part k of Fig. 5 illustrates that Tutor o $\rightarrow$ Lecturer and Tutor o $\rightarrow$ Professor. This means that a tutor is either a lecturer or a professor, but there might also be a tutor who is neither a professor nor a lecturer. On the other hand, if Tutor $\rightarrow$ Lecturer and Tutor $\rightarrow$ Professor, then every tutor participates in the FD, which means every tutor must be a lecturer or a professor.

element definition. Let Oid, $A_1, A_2, \ldots, A_n$ be the attributes of O. By Lemma 2, D has redundant data values with respect to a non-trivial FD $X \rightarrow A_i$ where $XA_i \subseteq \text{Oid}A_1A_2 \ldots A_n$ if and only if D has redundant data values with respect to Oid $\rightarrow A_i$. Hence, it is sufficient to focus on the FDs that have Oid as the left-hand sides. By Lemma 1, D has redundant data values with respect to Oid $\rightarrow A_i$ if and only if there are two distinct instances I_1 and I_2 of E that represent the same instance o of O and they have at least two common attributes and one of which is Oid. Therefore, we shall prove by induction on the number n of iterations of the repeat loop in Algorithm 1 that there are not two distinct element instances that represent the same object instance. As a result, D has no redundant attribute values. When $n = 0$, G has only the root XML element. Hence, any XML document over G can only have a single XML element instance, which means it cannot have any redundancy. Assume Algorithm 1 generates redundancy-free DTD graphs if the number of iterations of the repeat-loop in Algorithm 1 is less than or equal to an integer k where $k \geq 0$. Now consider the repeat loop executes the $(k+1)$th iteration. At the $(k+1)$th iteration, Algorithm 1 selects the remaining smallest XML elements in S. Let A be a selected XML element. If $k > 0$, there is an XML element B such that A becomes a child element of B and B was added to G in the kth iteration and $B^+ \supset A^+$ (or equivalently, $A \rightarrow B$.) If $k = 0$, then $A = A^+$. If $k > 0$, by the induction hypothesis there are not two distinct B element instances that represent the same object instance. Since $A \rightarrow B$, as a result there are not two distinct A element instances that represent the same object instance. If $k = 0$, A becomes a child element of the root element. As a result there are not two distinct A element instances that represent the same object instance. The induction step is now complete. □

Theorem 2. *Algorithm* 1 *generates a minimum number of fragments.*

Proof. Let H be the input semantic metadata and G be the output DTD graph. By Definition 1, we need to prove that the root element has the minimum possible number of children. Otherwise, we will have data redundancy or missing information. At the beginning, Algorithm 1 adds every XML element with the smallest closure as a child of the root element. For any two distinct smallest XML elements E_A and E_B, if $E_A \nrightarrow E_B$, E_A cannot be a child of E_B. On the other hand, if $E_A \rightarrow E_B$ but the FD is optional on the tail side, E_A cannot be a child of E_B either. Since E_A is optional in the FD, there may be some E_A object instances not related to any E_B instances. As a result, those E_A object instances that do not participate in the FD cannot be represented by any E_A element instances because they do not have parents. The argument that E_B cannot be a child of E_A is similar. Hence, both E_A and E_B must be children of the root element. For the special case that if two smallest elements E_A and E_B such that $E_A^+ = E_B^+$, the algorithm adds one of the two, say E_A, as a child of the root element. E_B will become a child of E_A later. All other elements in S can be added as a child element of other XML elements

other than the root element of G. Hence, Algorithm 1 will not add anymore fragment to G. □

Theorem 3. *Algorithm* 1 *runs in polynomial time.*

Proof. After extracting the FDs from the users' requirements, generating the functional closure for an object type takes linear time in terms of the number of FDs.[8] Given $n \geq 1$ input object types, the number of extracted FDs are bounded by $n(n-1)/2$. Thus, generating the functional closure for an object type takes at worst n^2 time. As a result, generating n functional closures for n input object types is thus bounded by n^3 time. The rest of the algorithm consists of comparing n functional closures and building the output DTD graph. At most $n(n-1)/2$ comparisons are needed for the n functional closures and each comparison at worst requires scanning n object types. Thus, comparing the n functional closures at worst takes n^3 time. Building the output DTD graph clearly needs less than n^3 time. Thus, the greatest factor is n^3, which implies Algorithm 1 runs in polynomial time. □

5. An Extensive Example

This section presents an extensive example. The input object types and their relationships are shown as follows:

(a) An employee may be a spouse of another employee (a unary and a one-to-one relationship: Employee o → Spouse[e] and Spouse → Employee).
(b) A tutor may help more than one student for more than one course (a ternary relationship).
(c) A tutor can be a lecturer or a professor or something else (Tutor o → Lecturer and Tutor o → Professor).
(d) A department has only one department head and a department head can only chair one department (a one-to-one relationship: Head → Department and Department → Head). A department head is an employee, but an employee may not be a department head (Head → Employee).
(e) An instructor is an employee but an employee may not be an instructor (Instructor → Employee).
(f) An instructor must belong to a department (a many-to-one relationship: Instructor → Department).
(g) An instructor can either be full-time or part-time (FullTime → Instructor and PartTime → Instructor).
(h) An instructor can teach many courses and a course can be taught by many instructors (a many-to-many relationship).
(i) A full-time instructor has a retirement plan (a many-to-one relationship: FullTime → RetirementPlan).

[e]There may be some employees who do not have spouses.

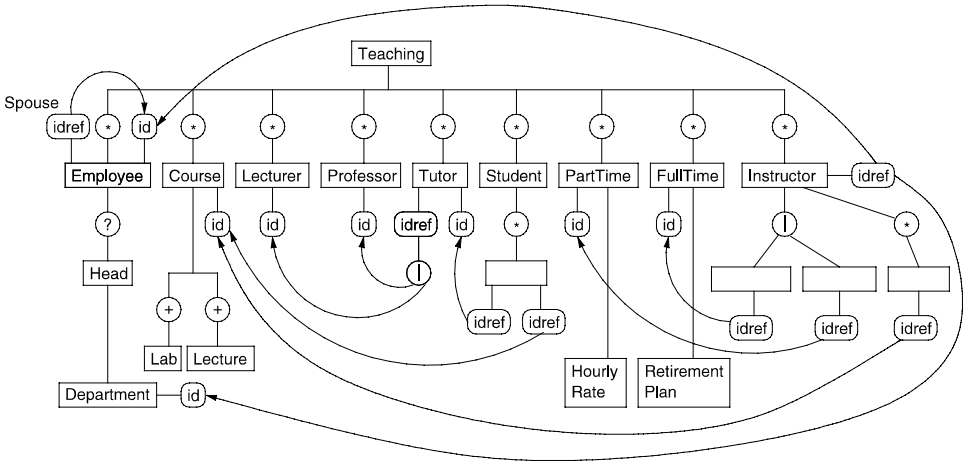


Fig. 9. A "bad" DTD-graph that has too many fragments.

(j) A part-time instructor has an hourly rate (a many-to-one relationship: PartTime → HourlyRate).
(k) A course consists of lectures and laboratories (two many-to-one relationships: Lecture → Course and Laboratory → Course).

A "bad" design that has too many fragments is shown in Fig. 9. Particularly, the DTD graph in Fig. 9 does not take advantage of the relationship between the object types Instructor and Employee, which yields the FD Instructor → Employee. Because of this FD, Instructor can be nested as a child XML element of Employee, which would result in one less fragment. On the other hand, Algorithm 1 recognizes this FD and will make Instructor a child element of Employee. We now trace through Algorithm 1. By the extracted FDs, we generate the following functional closures of the XML elements:

Student$^+$ = Student,
Tutor$^+$ = Tutor,
Lecturer$^+$ = Lecturer,
Professor$^+$ = Professor,
RetirementPlan$^+$ = RetirementPlan,
HourlyRate$^+$ = HourlyRate,
Course$^+$ = Course,
Employee$^+$ = Employee,

Instructor$^+$ = Instructor Employee Department,
Head$^+$ = Head Department Employee,
Department$^+$ = Department Head Employee,
Lab$^+$ = Lab Course,
Lecture$^+$ = Lecture Course,

$\text{FullTime}^+ = \text{FullTime RetirementPlan} \cup \text{Instructor}^+$,
$\text{PartTime}^+ = \text{PartTime HourlyRate} \cup \text{Instructor}^+$.

In the first pass of the repeat loop of Algorithm 1, the smallest XML elements are those that are equal to their functional closures. They are Student, Tutor, Lecturer, Professor, RetirementPlan, HourlyRate, Course, and Employee. They then become child XML elements of the root element. Algorithm 1 also adds the input data semantics of these XML elements to the graph. The resulting DTD graph is shown in Fig. 10. For the FD Spouse $\rightarrow$ Employee, Spouse is a role of the object type Employee. As such, it is better to model Spouse as an optional attribute that references the id field of the Employee XML element definition. It is obvious that if an employee does not have a spouse, the XML element instance that represents the employee does not have this optional attribute.

In the second pass of the repeat loop of Algorithm 1, the algorithm moves the remaining smallest XML elements in S to G. They are Instructor, Head, Department, Lab and Lecture. Since $\text{Employee} \in \text{Instructor}^+$, Instructor becomes a child XML element of Employee. Since $\text{Head}^+ = \text{Department}^+$, Algorithm 1 adds both Head and Department to G. After which, the algorithm makes Head a child XML element of Employee since $\text{Employee} \in \text{Head}^+$. Since $\text{Lab}^+ = \text{Lab Course}$ and $\text{Lecture}^+ = \text{Lecture Course}$, both Lab and Lecture become child XML elements of Course. Afterwards, the input data semantics of these XML elements are added to the graph. The resulting DTD graph is shown in Fig. 11.

In the third pass of the repeat loop of Algorithm 1, the algorithm moves the remaining smallest XML elements in S to G. They are FullTime and PartTime. Since $\text{Instructor} \in \text{FullTime}^+$ and $\text{Instructor} \in \text{PartTime}^+$, FullTime and PartTime become child XML elements of Instructor, as shown in Fig. 12. The input data semantics of these XML elements are added to the graph as well.

Note that the DTD graph in Fig. 12 takes advantage of the extracted FDs. Therefore, it does not have extra fragments. The number of fragments cannot be

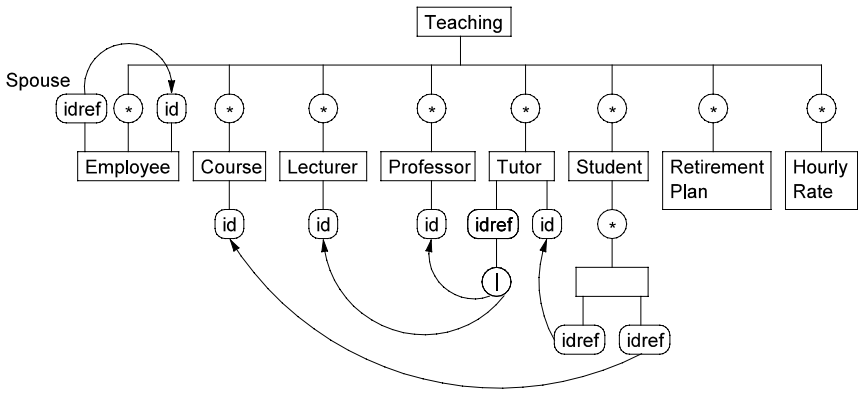


Fig. 10. The resulting DTD graph after the first pass of the repeat loop of Algorithm 1.

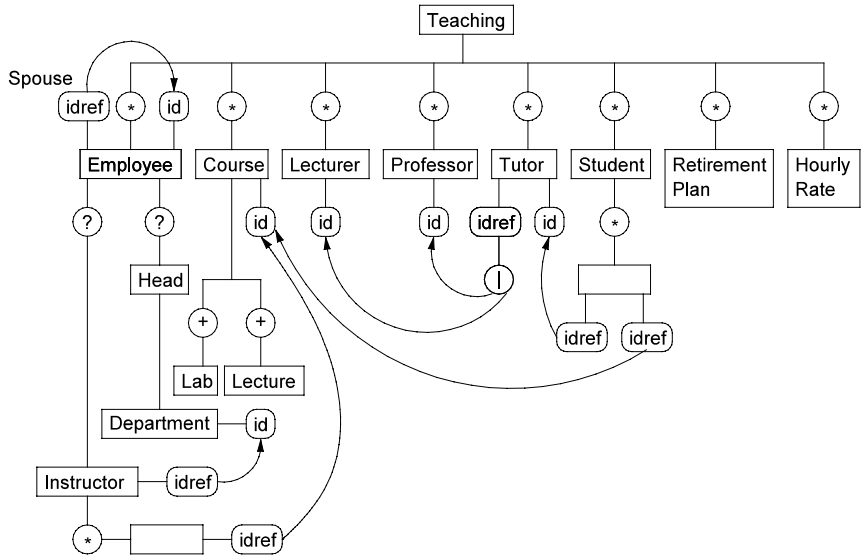


Fig. 11. The resulting DTD graph after the second pass of the repeat loop of Algorithm 1.

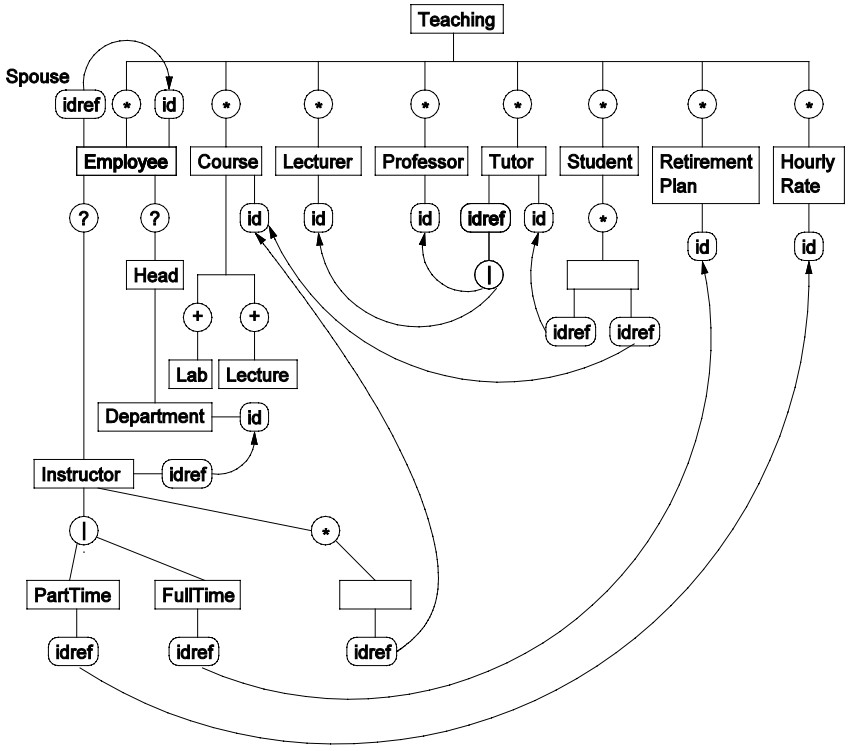


Fig. 12. The resulting DTD graph after the third pass of the repeat loop of Algorithm 1.

further reduced without introducing redundant data values or missing information. Thus, the DTD graph in Fig. 12 is a superior design to that in Fig. 9.

6. Performance Analysis

This section presents experimental data to substantiate our claim that the smallest closure approach indeed generates XML schemes that will reduce access time. Specifically, we generated XML files in accordance with the DTD graphs in Figs. 9 and 12 and executed two Microsoft LINQ queries in C#. Two sample XML files are shown in Figs. 13 and 14, respectively. As a comparison, an XML file with redundant RetirementPlan elements is also shown in Fig. 15. To facilitate the queries, we added an attribute "MartialStatus" to the XML element Employee, which is a common attribute of the object type Employee.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Teaching SYSTEM 'figure9.dtd'>
<Teaching>
  <Employee EID='E1' MaritalStatus='S'></Employee>
  <Employee EID='E2' MaritalStatus='M'></Employee>
.................
  <Course CID='C1'>
    <Lab>SAP Lab</Lab>
    <Lecture>Intro to SAP</Lecture>
  </Course>
.................
  <PartTime PID='P1'>
    <HourlyRate HID='H1'>$13.47</HourlyRate>
  </PartTime>
.................
  <FullTime FID='F1'>
    <RetirementPlan RID='R1'>Retirement age = 65, TIAA-CREF</RetirementPlan>
  </FullTime>
.................
  <Instructor IID='E1'>
    <PartTimeEmployment PID='P2'/>
    <CourseTaught CID='C9'/>
  </Instructor>
  <Instructor IID='E2'>
    <FullTimeEmployment FID='F3'/>
    <CourseTaught CID='C1'/>
    <CourseTaught CID='C3'/>
  </Instructor>
.................
</Teaching>
```

Fig. 13. An XML document that complies with the DTD graph in Fig. 9.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Teaching SYSTEM 'figure12.dtd'>
<Teaching>
  <Employee EID='E1' MaritalStatus='S'>
    <Instructor>
      <PartTime PID='H2'/>
      <CourseTaught CID='C8'/>
    </Instructor>
  </Employee>
  <Employee EID='E2' MaritalStatus='M'>
    <Instructor>
      <FullTime FID='R3'/>
    </Instructor>
  </Employee>
.....................
  <Course CID='C1'>
    <Lab>SAP Lab</Lab>
    <Lecture>Intro to SAP</Lecture>
  </Course>
.....................
  <RetirementPlan RID='R1'>Retirement age = 65, TIAA-CREF</RetirementPlan>
.....................
  <HourlyRate HID='H1'>$13.47</HourlyRate>
.....................
<Teaching>
```

Fig. 14. An XML document that complies with the DTD graph in Fig. 12.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Teaching SYSTEM 'figureRedundancy.dtd'>
<Teaching>
  <Employee EID='E1' MaritalStatus='S'>
    <Instructor>
      <PartTime PID='H2'/>
      <CourseTaught CID='C5'/>
    </Instructor>
  </Employee>
  <Employee EID='E2' MaritalStatus='M'>
    <Instructor>
      <FullTime>
        <RetirementPlan>Retirement age = 59, Fidelity Investments</RetirementPlan>
      </FullTime>
    </Instructor>
  </Employee>
.....................
  <Employee EID='E5' MaritalStatus='M'>
    <Instructor>
      <FullTime>
        <RetirementPlan>Retirement age = 59, Fidelity Investments</RetirementPlan>
      </FullTime>
      <CourseTaught CID='C1'/>
      <CourseTaught CID='C2'/>
      <CourseTaught CID='C8'/>
    </Instructor>
  </Employee>
```

Fig. 15. An XML document that has redundant RetirementPlan elements.

```
.....................
  <Course CID='C1'>
    <Lab>SAP Lab</Lab>
    <Lecture>Intro to SAP</Lecture>
  </Course>
.....................
  <HourlyRate HID='H1'>$13.47</HourlyRate>
.....................
</Teaching>
```

Fig. 15. (*Continued*)

The Microsoft LINQ queries used in the experiments are shown in the following tables:

Query 1: Find the retirement plan of every unmarried full-time instructor.

Fig. 9	var result = new XElement("results", from e in fig9.Element("Teaching").Elements("Employee") where (string)e.Attribute("MaritalStatus").Value == "S" from i in fig9.Element("Teaching").Elements("Instructor") where (string)i.Attribute("IID").Value == (string)e.Attribute("EID").Value from r in i.Elements("FullTimeEmployment") from f in fig9.Element("Teaching").Elements("FullTime") where (string)f.Attribute("FID").Value == (string)r.Attribute("FID") select new XElement("SelectedEmployee", new XAttribute("EID", (string)e.Attribute("EID").Value), f.Descendants("RetirementPlan")));
Fig. 12	var result = new XElement("results", from e in fig12.Element("Teaching").Elements("Employee") where (string)e.Attribute("MaritalStatus").Value == "S" from f in e.Elements("Instructor").Elements("FullTime") from r in fig12.Element("Teaching").Elements("RetirementPlan") where (string)f.Attribute("FID").Value == (string)r.Attribute("RID").Value select new XElement("SelectedEmployee", new XAttribute("EID", (string)e.Attribute("EID").Value), r));
A DTD that has Redundant Retirement Plan Elements	var result = new XElement("results", from e in figRedundancy.Element("Teaching").Elements("Employee") where (string)e.Attribute("MaritalStatus").Value == "S" from r in e.Elements("Instructor").Elements("FullTime").Elements("RetirementPlan") select new XElement("SelectedEmployee", new XAttribute("EID", (string)e.Attribute("EID").Value), r));

Query 2: Find the retirement plan of every full-time instructor.

Fig. 9	var result = new XElement("results", from i in fig9.Element("Teaching").Elements("Instructor") from r in i.Elements("FullTimeEmployment") from f in fig9.Element("Teaching").Elements("FullTime") where (string)f.Attribute("FID").Value == (string)r.Attribute("FID") select new XElement("SelectedEmployee", new XAttribute("IID", (string)i.Attribute("IID").Value), f.Descendants("RetirementPlan")));
Fig. 12	var result = new XElement("results", from f in fig12.Element("Teaching").Elements("Employee").Elements("Instructor").Elements("FullTime") from r in fig12.Element("Teaching").Elements("RetirementPlan") where (string)f.Attribute("FID").Value == (string)r.Attribute("RID").Value from e in f.Ancestors("Employee") select new XElement("SelectedEmployee", new XAttribute("EID", (string)e.Attribute("EID").Value), r));

A DTD that has Redundant Retirement Plan Elements	var result = new XElement("results", from r in figRedudnancy.Element("Teaching").Elements("Employee").Elements("Instructor").Element s("FullTime").Elements("RetirementPlan") from e in r.Ancestors("Employee") select new XElement("SelectedEmployee", new XAttribute("EID", (string)e.Attribute("EID").Value), r));

The queries were executed on a Windows 8.1 64-bit desktop computer that has an AMD A6-3650 CPU with a Radeon^{TM} HD Graphics processor and 8 GB memory. In all the XML files used in the experiments, the number of Employee elements exceeded the number of Instructor elements, simulating the fact that there are employees who are not instructors. In addition, all XML files used in the experiments have ten Course elements, five HourlyRate elements, and three RetirementPlan elements. Each Instructor element is randomly assigned 0 to 4 Course elements, simulating that each instructor may teach up to 4 courses. Further, every part-time Instructor element is followed by nine full-time Instructor elements and every unmarried Employee element is followed by six married Employee elements. We executed the queries with 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 Instructor elements. The execution time of each query was measured in terms of ticks, the smallest unit of time that the C# Stopwatch timer can measure. For each LINQ query of a DTD and for each number of Instructor elements, we executed the query 53 times, discarded the first three executions, and averaged the results of the last fifty executions. We noted that the first several executions took much longer than the other executions. Therefore, we discarded the first three "warm-up" executions and averaged the last fifty executions. The execution times measured in ticks are shown in Table 1.

Figures 16 and 17 plot the execution times of Queries 1 and 2 against the number of Instructor elements.

It is expected that LINQ will spend more time to execute Query 1 on XML files that comply with the DTD graph in Fig. 9. From the LINQ code we can see that the unmarried Employee elements are first selected. Then, LINQ selects the matching Instructor elements. LINQ then selects the unmarried Instructor elements that

Table 1. Execution times of the LINQ queries.

Num. of Instr. Elements	Query 1			Query 2		
	Fig. 9	Fig. 12	With Redundancy	Fig. 9	Fig. 12	With Redundancy
200	36587.32	3311.56	1368.64	23958.62	16039.38	3771.46
400	131540.22	8999.2	2568.58	77312.2	49351.54	7749.14
600	288776.18	16721.74	4668.86	164508.12	99925.14	11294.64
800	513848.44	26713.54	5254.34	263681.74	164371.62	14691.46
1000	795169.48	38973.06	6335.14	398863.78	250284.58	17391.22
1200	1167544.14	54981.5	8028.02	564378.5	350924.98	22728.86
1400	1523601.94	71058.12	8976.38	761585.3	459157.86	24131.24
1600	2004435.86	91389.62	11907.42	948968.58	591074.54	27669.04
1800	2481122	111971.86	13929.16	1212540.84	734541.92	30704.94
2000	3084158.7	137819.64	14438.2	1475694.64	901096.28	34677.16

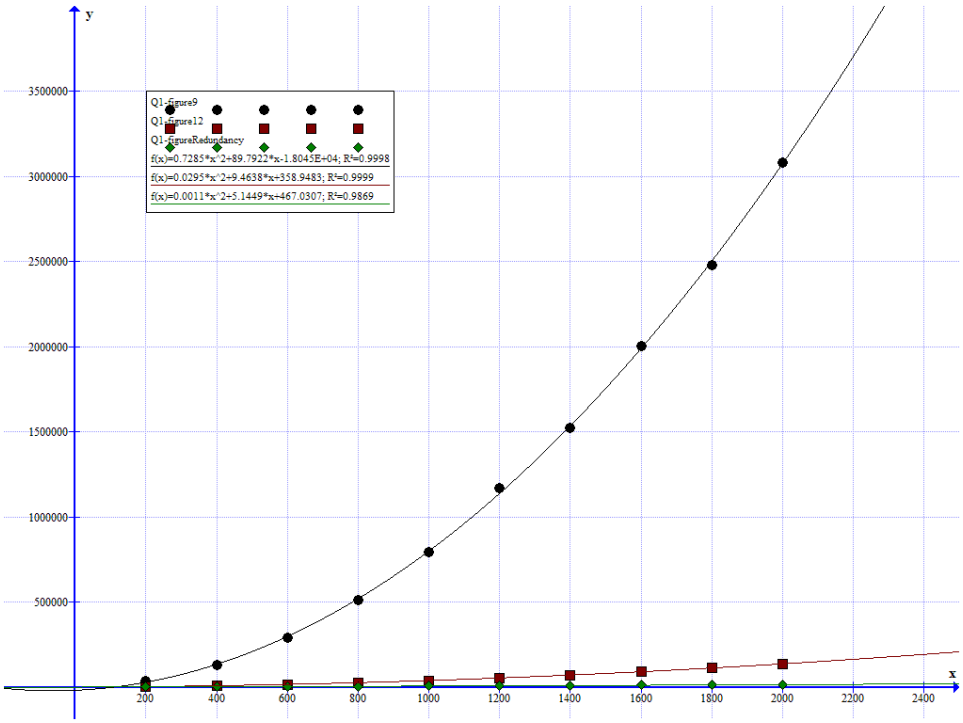


Fig. 16. Execution time of Query 1 against the number of Instructor elements.

have FullTimeEmployment child elements. It then selects the matching FullTime elements. Finally, it selects the RetirementPlan child elements of the matched FullTime elements. For the XML files that comply with the DTD graph in Fig. 12, LINQ again selects the unmarried Employee elements first. However, because Instructor elements are nested within Employee elements, LINQ does not need to go to another fragment to find the matching Instructor elements. For each unmarried Instructor element, LINQ next selects the RetirementPlan ID of the FullTime element nested within the Instructor element. Finally, LINQ finds the matching RetirementPlan element. For the XML files that have redundant RetirementPlan elements, LINQ first needs to select the unmarried Employee elements. Then, no matching needs to be done because the RetirementPlan elements are nested within the unmarried FullTime elements.

Query 2 is similar to Query 1, but it requires one less matching. For the XML files that comply with the DTD graph of Fig. 9, LINQ first goes from the Teaching element, then each Instructor element, then the FullTimeEmployment element within the Instructor element, and then LINQ goes from the Teaching element, then the matching FullTime element and its RetirementPlan child element. The number of elements traversed for each instructor is thus $3 + 3 = 6$. For the XML files that comply with the DTD graph of Fig. 12, LINQ first goes from the Teaching element,

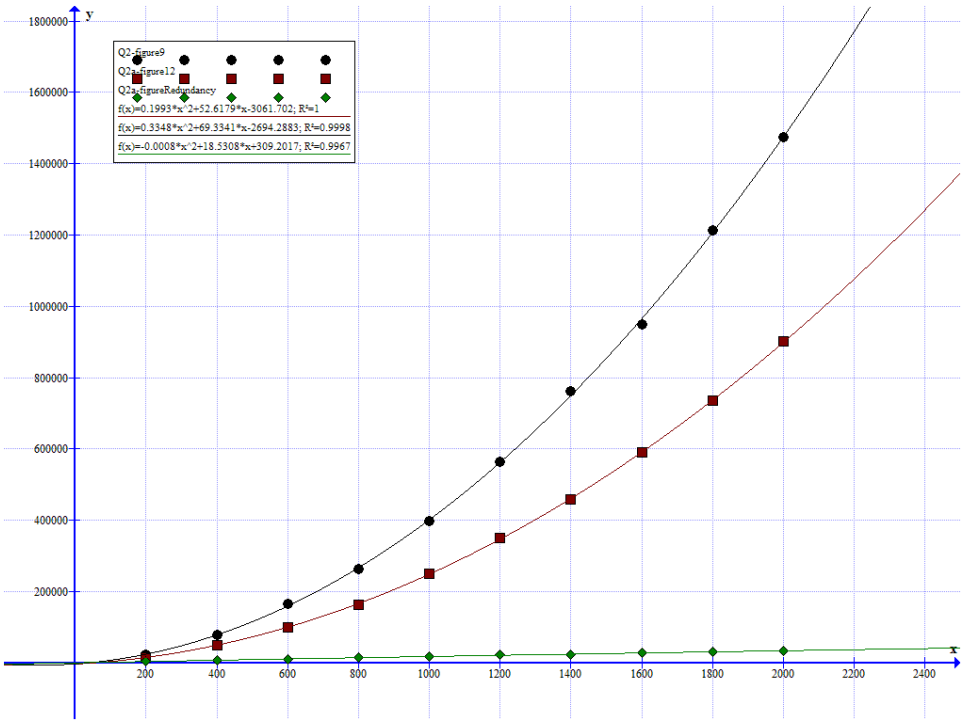


Fig. 17. Execution time of Query 2 against the number of Instructor elements.

then each Instructor element, then its FullTime element, and then LINQ goes from the Teaching element, and to the matching RetirementPlan element. The number of elements traversed for each instructor is thus $4 + 2 = 6$. Although the number of elements traversed for each instructor is the same in both cases, as far as Query 2 is concerned the XML files that comply with the DTD graph of Fig. 12 still outperforms than those that comply with the DTD graph of Fig. 9. With redundant RetirementPlan elements, the number of elements traversed for each instructor is only 5.

Because LINQ does not traverse a fragment that consists of RetirementPlan elements, redundant RetirementPlan elements speed up Queries 1 and 2. However, the price for fast access is that the size of the XML file also increases. Measured in bytes, Table 2 presents the size of the XML file used in the experiments. Using the data in Table 2, Fig. 18 plots the size of the XML file against the number of Instructor elements, which shows that the size of the XML file increases linearly with the number of Instructor elements.

Although storage cost has dropped to almost zero today, response time is still a critical matter. In our experiments, the LINQ queries retrieve RetirementPlan elements. Therefore, redundant RetirementPlan elements readily speed up the queries. In fact, it is well known that if the access pattern of the user is known beforehand

Table 2. XML file size (in bytes) against the number of Instructor elements.

Num. of Instr. Elements	Fig. 9	Fig. 12	With Redundancy
200	34918	30715	44249
400	68144	60591	87306
600	103052	89133	129846
800	135707	118391	173052
1000	168663	146444	212982
1200	202557	175632	256988
1400	236461	203351	298445
1600	266856	233424	344776
1800	301648	262250	382765
2000	334587	293264	427354

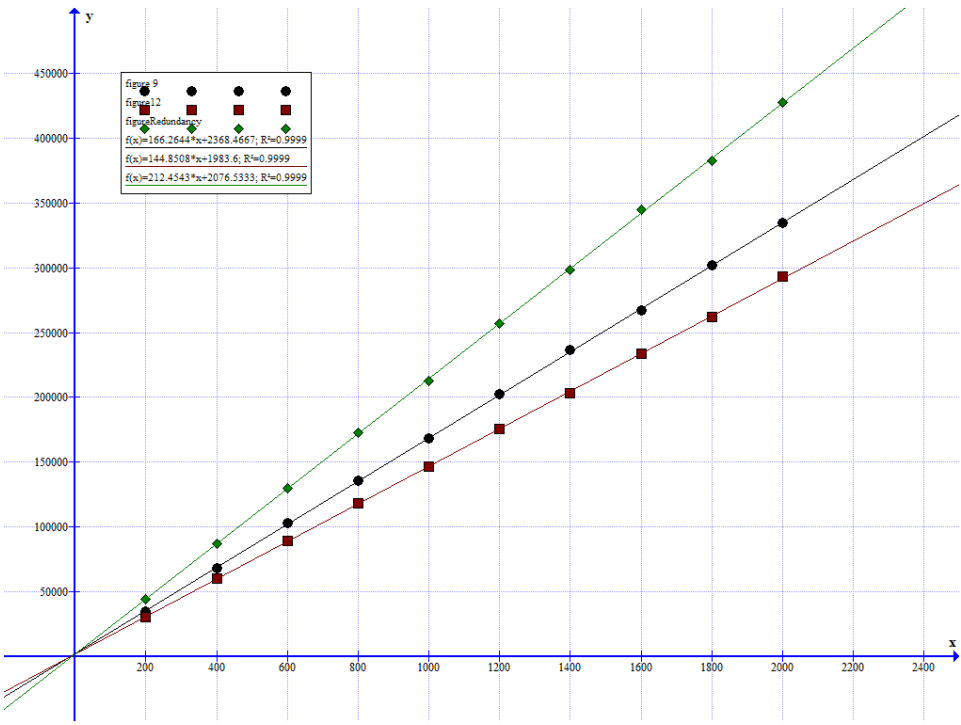


Fig. 18. XML file size (in bytes) against the number of Instructor elements.

and update on redundant data is relatively few, redundancy can improve access time. However, if the access pattern of the data is unknown and update on redundant data is frequent, redundancy may increase response time. Therefore, whether or not redundancy will improve access time depends on the actual access pattern and the frequency of the update on the redundant data. On the other hand, the smallest closure approach of this paper does not depend on the access pattern of the data. It relies on the relationships of the input object types. The output of our approach has the minimum number of fragments and the complying XML files

do not have redundant data. In fact, traversing the extra fragments significantly increases access time, as shown in Fig. 16.

Conversely, given the data semantics of Sec. 5, our algorithm will correct the redundancy problem of the XML file in Fig. 15. The main reason is that the functional closure of RetirementPlan is equal to itself, i.e. RetirementPlan$^+$ = RetirementPlan. Because its functional closure is equal to itself, RetirementPlan will become a child element of the root element. Hence, the smallest closure approach of this paper will not generate a DTD that allows the XML file in Fig. 15. In fact, the smallest closure approach of this paper will generate the DTD graph of Fig. 12, and its complying XML files are the smallest in our experiments.

To close this section, we note that (i) DTDs with unnecessary fragments will make complying XML files to have long access time, especially if queries need to traverse unnecessary fragments to retrieve data, and (ii) using redundant RetirementPlan elements to speed up queries comes with a price. If the access pattern of the data is unknown and update on redundant data is frequent, redundant RetirementPlan elements may actually slow down response time. In fact, using redundant data to speed up queries is very application specific.

7. Conclusion

This paper proposed the use DTD graphs and XSD graphs as a mean for conceptual modeling of XML applications. Because DTD graphs and XSD graphs are trees, they can easily be mapped to DTDs and XSDs for implementation. This paper also defined a set of transformation rules that can be used to transform DTDs to a large class of XML schemas and vice versa. Then, this paper provided an algorithm that inputs users' requirements and outputs a DTD graph that has a minimum number of redundancy-free fragments. As a result of these good properties, the resulting DTD or XSD facilitates query processing and update. In other words, minimum fragments in XML schema design can reduce query time, that is, less time to navigate through pointers between fragments. Furthermore, redundancy free XML schema design can reduce update time, that is, less time to update a non-redundant element in XML document.

References

1. Y. Chen, S. B. Davidson, C. S. Hara and Y. Zheng, RRXF: Redundancy reducing XML storage in relations, in *Proc. 29th Int. Conf. Very Large Data Bases*, pp. 189–200, Berlin, Germany, September 9–12, 2003.
2. E. F. Codd, Recent investigations in relational data base systems, in *IFIP Congr.*, pp. 1017–1021.
3. J. Fong, *Information Systems Reengineering and Integration* (Springer, 1974).
4. J. Fong, S. Cheung and H. Shiu, The XML tree model: Toward an XML conceptual schema reversed from XML schema definition, *Data Know. Eng.* **64**(3) (2008) 624–661.

5. J. E. Funderburk, G. Kiernan and J. Shanmugasundaram, Technical note XTABLES: Bridging relational technology and XML, *IBM Syst. J.* **42**(3) (2003) 538–541.
6. J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita and C. Wei, XTABLES: Bridging relational technology and XML, *IBM Syst. J.* **41**(4) (2002) 616–641.
7. L. Libkin, Normalization theory for XML, in *Proc. 5th Int. XML Database Symp.*, pp. 1–13, Vienna, Austria, September 23–24, 2007.
8. D. Maier, *The Theory of Relational Databases* (Computer Science Press, Maryland).
9. W. Y. Mok and D. W. Embley, Generating compact redundancy-free XML documents from conceptual-model hypergraphs, *IEEE Trans. Knowl. Data Eng.* **18**(8) (2006) 1082–1096.
10. W. Y. Mok, J. Fong and D. W. Embley, Extracting a largest redundancy-free XML storage structure from an acyclic hypergraph in polynomial time, *Inf. Syst.* **35**(7) (2010) 804–824.
11. W. Y. Mok, J. Fong and D. W. Embley, Generating the fewest redundancy-free scheme trees from acyclic conceptual-model hypergraphs in polynomial time, *Inf. Syst.* **41** (2014) 20–44.
12. K. D. Schewe, Redundancy, dependencies and normal forms for XML databases, in *Proc. Sixteenth Australasian Database Conf.*, pp. 7–16, Newcastle, Australia, 31st January–3rd February, 2005.
13. M. W. Vincent, J. Liu and C. Liu, Strong functional dependencies and their application to normal forms in XML, *ACM Trans. Database Syst.* **29**(3) (2004) 445–462.
14. w3 schools.com (Accessed on 30 October 2014). Introduction to XML Schema, http://www.w3schools.com/schema/schema_intro.asp.
15. J. Wang and R. W. Topor, Removing XML data redundancies using functional and equality-generating dependencies, in *Proc. Sixteenth Australasian Database Conf.*, pp. 65–74, Newcastle, Australia, January 31st–3rd February, 2005.
16. C. Yu and H. V. Jagadish, XML schema refinement through redundancy detection and normalization, *VLDB J.* **17**(2) (2008) 203–223.