XLearner: A System to Learn XML Queries from Examples

Atsuyuki Morishima
Akira Matsumoto
Hiroyuki Kitagawa

March 26, 2004

ISE-TR-04-196

Institute of Information Sciences and Electronics
University of Tsukuba
Tennohdai, Tsukuba, Ibaraki 305-8573, Japan

XLearner: A System to Learn XML Queries from Examples

Atsuyuki Morishima (mori@slis.tsukuba.ac.jp)
Akira Matsumoto (akey@kde.is.tsukuba.ac.jp)
Hiroyuki Kitagawa (kitagawa@is.tsukuba.ac.jp)

Abstract

This paper presents XLearner, a novel tool that helps the rapid development of XML mapping queries written in XQuery. XLearner is novel in that it learns XQuery queries consistent with given examples (fragments) of intended query results. XLearner combines known learning techniques, incorporates mechanisms to cope with issues specific to the XQuery learning context, and provides a systematic way for the semi-automatic development of queries. This paper describes the XLearner system. It presents algorithms for learning various classes of XQuery, shows that a minor extension gives the system a practical expressive power, and reports experimental results to demonstrate how XLearner outputs reasonably complicated queries with only a small number of interactions with the user.

1 Introduction

As the amount of XML data grows, the need to integrate and restructure XML data increases significantly. A major approach is to write XML queries to map one XML data structure into another, but developing such mapping queries requires tremendous effort. The effort is inevitable because of XML data characteristics and the nature of integration/restructuring: (1) XML data is nested, unnormalized, and an instance of semistructured data. This requires complex navigation and pattern matching of XML data. (2) Mapping XML data to a given complex schema requires complex construction of XML query results.

Since the development requires tremendous effort, tools to assist query development are beneficial. A well-known example is the theoretical tools for type checking [4], which checks if every output from an XML query is consistent with a given DTD. Another example is rapid development tools for XML mapping queries [16]. The latter tools are intended to play a similar role to rapid application development tools for programming languages; i.e., they are designed to help developers make their development processes more efficient. In general, such RAD tools provide various ways for the semi-automatic generation of code.

Our proposed tool, called *XLearner*, is a rapid query development tool. A key feature is that it uses machine learning techniques to generate *complicated* XQuery

queries after a small number of *simple* interactions with the user. XLearner requires the user merely to (1) show the system examples (fragments) of the intended query results and (2) answer simple Yes or No questions posed by the system. In other words, XLearner *learns* XQuery queries if the user gives example fragments and answers to the simple questions posed by XLearner. It is worth noting that the questions do not involve query language jargon such as joins.

Development of XLearner is challenging for three reasons. First, learning XML queries through examples is anything but trivial: XML is an instance of semistructured data, whose data structure is often irregular and implicit. This is a striking difference from relational databases, where the data structure is completely regular and explicit. The structure of XML query results is deeply nested and complex in general, in contrast to relational databases where the result is always a flat table. XLearner combines known machine learning techniques, incorporates mechanisms to cope with issues specific to the XQuery context, and provides a *non-adhoc framework* to achieve such non-trivial tasks. We believe this is a significant contribution.

Second, naive approaches have problems with computational complexity. Many XML query languages support path regular expressions or their variations, such as XPath [21]. The expressive power of regular expressions is equivalent to that of finite automata[1], and Gold has shown that the problem of finding a dfa of a minimum number of states compatible with a given set of examples is NP-complete [9]. It is known, however, that *active learning* [3] gives a polynomial-time algorithm to solve the problem, where active learning is a framework that allows the system (learner) to ask the user (teacher) questions. XLearner applies an active learning algorithm to our query learning context.

Finally, even with a polynomial-time algorithm, a tool is not necessarily practical. XLearner is based on an active learning framework, and requires interactions with the user. Keeping the number of interactions between XLearner and the user small is a critical point. If XLearner required several hundred interactions to learn a simple query, it would not be a practical system. XLearner reduces the number of interactions in the following way: (1) It uses constraints specific to XML queries. (2) It requires the input of explicit query specifications in part. Specifically, the user is required to specify selection predicates on values. Because users usually know what their predicates are and teaching such predicates by example is more troublesome than specifying them, this provides a balanced combination for the efficient development of queries.

Development of XLearner also presents an interesting problem: "which class of XQuery query is learnable through examples?" We give algorithms for *MAT-learning* various classes of XQuery queries, which means that queries in the classes are learnable if the user is a *minimally adequate teacher* [2]. A minimally adequate teacher answers particular types of questions. In addition, we propose a minor extension that gives the system a practical expressive power to learn queries, including 19 of 20 queries in XMark [18] benchmark and 11 of 12 queries in XML Use Case [20] "XMP" (Experiences and Exemplars).

In summary, the contribution of this paper is as follows: (1) We propose XLearner, an intelligent tool for learning XQuery queries through given examples of query results.

[1]The expressive power of XPath is not equal to, but overlaps that of regular expressions.

(2) We explain algorithms for learning various classes of XQuery queries in the active learning approach. (3) We show that a minor extension of the algorithm has practical expressive power. (4) We give experimental results to demonstrate that XLearner learns XQuery queries with a small number of interactions with the user.

Related Work. LSD [8] is a system that uses and extends machine learning techniques to semi-automatically create semantic mappings between a mediated schema and local schemas of data sources. It takes as input manual mappings between the mediated schema and a small set of data sources, and uses those mappings and data instances to propose mappings for subsequent data sources. With that capability, LSD is useful in a scenario where a data *integrator* plays a crucial role, who knows (1) how other data sources are mapped to the mediated schema and (2) what data is stored in the already-mapped data sources. On the other hand, XLearner requires no such information and uses other kinds of inputs. We believe XLearner is useful in another common scenario where each data *provider* plays a crucial role, who has to map their data to a given common schema.

Clio [16] is a well-known tool for the semi-automatic generation of schema-mapping queries written in SQL and XQuery. While XLearner and Clio look similar, they have different assumptions and different mechanisms to achieve the same goal. We believe future query development tools benefit from both approaches. Clio's code generation is based on *value-correspondences* given by the user and *semantic constraints* given by schemas. Clio generates query candidates by taking advantage of the assumption that the result of the generated query retains semantic constraints given by the source schema. XLearner uses machine learning techniques to *learn* queries from interactions with the user, and there is no default assumption on how to generate XML instances for the given target schema. Moreover, Clio assumes that the source XML has a regular structure, in the sense that generated queries contain only simple location paths (such as `/a/b/c`). This is a significant difference; XLearner can generate xpath expressions corresponding to regular languages, which contributes to the ability to learn a large class of queries.

There are theoretical studies in the field of computational learning theory. Angluin [2] introduced the concept of active learning and showed that a dfa having the minimum number of states consistent with given examples is learnable in polynomial time. Inspired by Angluin's work, researchers have been developing algorithms to learn a variety of patterns from data instances. Their focus is on learnability in polynomial time. Target patterns include different classes of formal languages [17][10] and subclasses of CNF and DNF [7]. Some works are on extracting patterns from trees or semistructured data [1][11][5]. In general, studies in this area are purely theoretical and cover only simplified problems; the development of XLearner is not a variation of them. The reason is two-fold: First, learning XQuery queries implies learning compositions of various constructs of queries, including path expressions, join/selection conditions, and nested queries. Second, as mentioned, polynomial-time learning is not a sufficient condition for interactive systems. As far as we know, XLearner is the first system designed for learning practical XML queries from given examples.

2 XLearner

For an example scenario, we use XML data taken from XMark [18] whose structure is modeled after a database deployed by an Internet auction site. Figure 1(a) shows a fragment of its DTD. An `item` is an object that is for sale or has already been sold. Each item has a unique identifier and properties like name and description, all encoded as elements. Each item is assigned a world region represented by the item's parent. A `category` has a unique identifier and a name. The categories are used to implement the classification of items, with `incategory` elements having IDREFs to the `category` elements. A `closed_auction` represents an auction that is finished. Its properties include a reference to the item sold and the price. Figure 4(a) shows a fragment of the data instance.

Consider the XQuery query q_1 shown in Figure 2 (ignore the superscripts). The query outputs an XML instance that conforms to the schema shown in Figure 1(b), where each `category` element contains a set of `items` whose world regions are `africa` or `europe` and that were sold for less than 300 dollars.

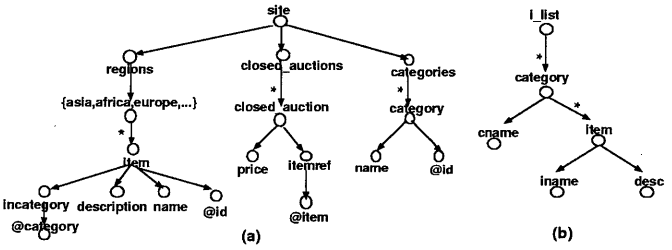


Figure 1: Schemas for an Example Scenario

XLearner. Figure 3 shows the XLearner's architecture. First, schema information (such as a DTD) of the query result is given to the template generator. Next, the user chooses several XML elements and values (We give them a generic name *XML node*) in target XML data, and drags and drops them into the generated template to show the system a fragment of the user's intended query result. We refer to the XML nodes dropped in the template as *dropped example nodes*. A space into which the user can drop an XML node is called a *Drop Box*.

According to the given query result fragment, the learning engine asks the user questions to infer the intended query. So the user is often called the *teacher*. Finally, the learning engine outputs an XQuery query.

We use the example scenario given earlier to illustrate those steps in more detail. First, the template generator receives the schema in Figure 1(b) and presents the template shown in Figure 4(b). The user then chooses several XML elements from the XML data shown in the XML browser (Figure 4(a)), and drags and drops them into the template so that the template and dropped nodes constitute a fragment of the intended query result. There is no constraint in constructing the fragment, except that the fragment must be consistent with the query result. We assume here that the fragment is given as shown in Figure 4, where an arrow indicates a drag-and-drop operation. Note

```
(N1)<i_list> {
 (N1.1)for $c in /site/categories/category
 return <category><cname>{(N1.1.1)$c/name}</> {
   (N1.1.2)for $i in /site/regions/(europe|africa)/item
   where $i/incategory/@category = $c/@id
     and some $o in /site/closed_auctions/closed_auction
     satisfies $o/itemref/@item = $i/@id and
     data($o/price)<300
   return <item><iname>{(N1.1.2.1)$i/name}</>
    {(N1.1.2.2) for $id in $i/description return <desc>$id</>}
   </>
 }</>
}</>
```

Figure 2: q_1 (`document()` and `text()` are omitted)

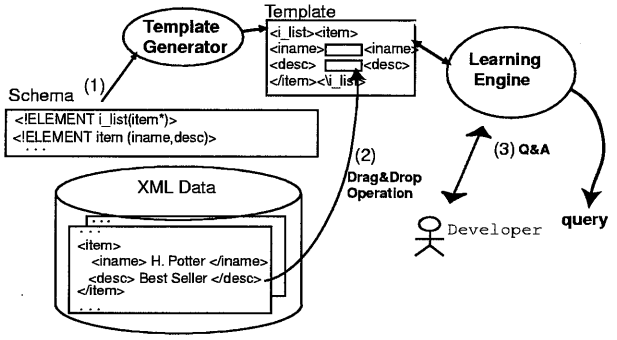


Figure 3: How to use XLearner

that "H. Potter" is an example of an item whose region is `europe` and was sold for less than 300 dollars.

XLearner then asks the user questions (called *learner's queries*, or just *queries* if there is no ambiguity) to identify the intended XQuery query (i.e., q_1 in Figure 2). More specifically, XLearner makes queries to identify the *extent* of each dropped example node e. Intuitively, the extent of e is the set of nodes represented by e. The most simple example is the extent of `book`, which is the set of all the category name nodes. Actually, the extent of e to be identified depends not only on e but also on a given *context*, denoted by $context(e)$. Intuitively, $context(e)$ is the set of dropped example nodes that may be influencial in identifying the extent of e. Assume that $context(\texttt{H.Potter}) = \{\text{"book"}\}$ is given. Then, the extent of "H. Potter" must be the set of item names represented by the "H. Potter" under the condition that the `cname` element contains "book." If XLearner is used to learn q_1, the extent should contain only names of items whose category is "book." Given e and $context(e)$, we write $EXT_{e,context(e)}$ to denote the extent of e in $context(e)$.

In the learning process, XLearner traverses dropped example nodes to identify their extents in the order that the user can consider $context(e)$ as the set of other dropped example nodes whose extents have already been identified. In the scenario, the extents are identified in the order of "book," "H. Potter," and "Best Seller." First, $EXT_{\texttt{book},\phi}$ has an empty context, and has to be identified through the answers to the learner's queries as the set of all the category name nodes. The extent of the second example ($EXT_{\texttt{H.Potter},\{\texttt{book}\}}$) has to be identified as the set of names of items, in its context (namely, for the given category name, i.e., "book"), whose regions are `africa` or `europe` and that were sold for less than 300 dollars. Finally, $EXT_{\texttt{BestSeller},\{\texttt{book},\texttt{H.Potter}\}}$ has to be identified as the set of descriptions of an item having the given item name (i.e., "H. Potter") in the given category name (i.e., "book"). We omit $context(e)$ in discussions unless it is important.

There are two types of learner's queries: *membership queries* and *equivalence queries.*

Membership Queries. To make a membership query for example e, XLearner chooses any XML node n and asks the user whether n is included in EXT_e. The user answers Yes (Y) or No (N). Figure 5 (a) shows an example of a membership query, which asks the user if the "XML book" under the `asia` element is included in $EXT_{\texttt{H.Potter}}$. The answer is N, because its region is neither `europe` nor `africa`. If $n \in EXT_e$ ($n \notin EXT_e$), n is called a *positive (negative) example.* Dropped examples are positive examples by definition.

Equivalence Queries. The queries use *hypotheses*, i.e., intermediate results in the learning process. We use the hat notation $\hat{x}$ to differentiate a hypothesis from the correct answer x to learn. To make an equivalence query, XLearner highlights XML nodes in a hypothesis extent $E\hat{X}T_e$ in the XML browser and asks if $E\hat{X}T_e = EXT_e$. If true, the user clicks on the [OK] button. If not, the user gives a *counterexample* ce to XLearner. A *counterexample* is an XML node in the symmetric difference between EXT_e and $E\hat{X}T_e$. If $ce \in EXT_e - E\hat{X}T_e$ ($ce \in E\hat{X}T_e - EXT_e$), ce is called a *positive (negative) counterexample.* A positive (negative) counterexample is also a positive (negative) example by definition. Figure 5 (b) shows an example of an equivalence query, where $E\hat{X}T_{\texttt{H.Potter}}$ is highlighted in the browser. The user says "Encyclopedia" (with `id` "i6") is a negative counterexample, because the sales price was more than 300 dollars.

A teacher with the ability to give answers to the two types of queries is called a *minimally adequate teacher* [2].

Occasionally XLearner presents a dialog box (called a *Condition Box*) and requires explicit conditions. In the scenario, XLearner gives a Condition Box in the process of identifying $EXT_{\texttt{H.Potter}}$. The user drags and drops H. Potter's price value (that is under `closed_auction` element) into the Condition Box and enters the selection condition `<300` (Figure 5)(c).

If unable to find appropriate examples in a context, the user can change the context by switching to other choices of dropped examples to specify the same query. Details are explained in Section 4.1.

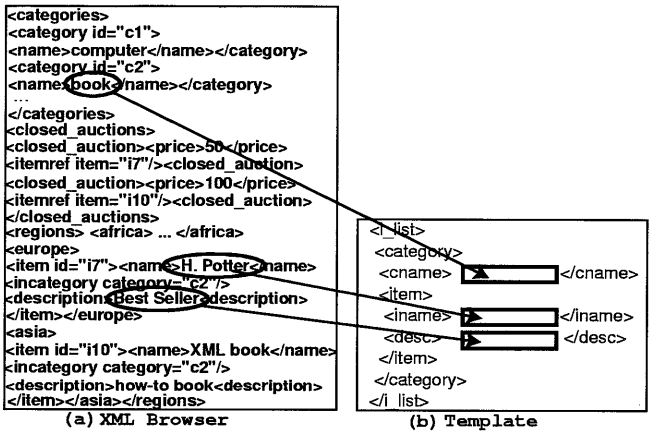


Figure 4: XML Browser and Template

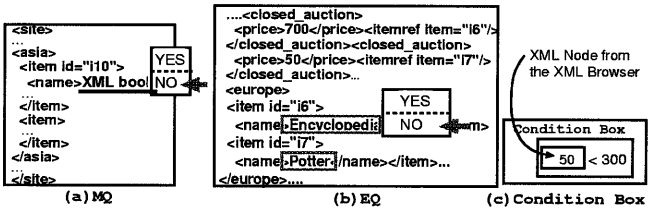
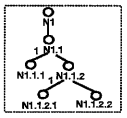


Figure 5: Interactions with XLearner

3 XQ-Tree

To discuss the learning algorithm, we introduce the XQ-Tree, our representation of XQuery queries. Figure 6 shows an XQ-Tree t_1 corresponding to q_1 in Figure 2. t_1 is essentially the same as q_1 except that (1) the nesting structure of `flwr` expressions is represented as a tree, and (2) it transforms every expression that returns a sequence of nodes (e.g. `$c/name`) into the equivalent `flwr` expression (`for $cn in $c/name return $cn`). Each node n in an XQ-Tree has the form Ni:-$q(Ni)$ where Ni is a *node identifier* (We use Dewey encoding [15] here) and $q(Ni)$ is a *query fragment* (in the form of flwr expression, [`for` $expr_f$ [`where` $expr_w$]] `return` $expr_r$, where [...] is optional). Tags and node identifiers in $expr_r$ are omitted unless they are important.

The XQ-Tree representation has two important properties: (1) Every expression that returns a sequence of XML nodes (i.e., `flwr` expression) has a variable over the nodes in the sequence. (2) Edges between nodes suggest *query dependencies* among the nodes. For example, evaluation of $q(n)$ depends on that of query fragments of n's ancestors ($ancestors_t(n)$) in general. We write $depends_t(n)$ to denote the set of all the nodes in t that n may depend on. As explained later, $depends_t(n)$ varies according to *classes* of queries. For the class to which the query in Figure 6 belongs,




```
N1:-        return <i_list>{N1.1}</i_list>
N1.1:-      for $c in /site/categories/category
            return <category><cname>{N1.1.1}</>{N1.1.2}</>
N1.1.1:-    for $cn in $c/name return $cn
N1.1.2:-    for $i in /site/regions/(europe|africa)/item
            where some $ic in $i/incategory/@category,
                  $ci in $c/@id satisfies ($ic=$ci)
              and some $o in
                  /site/closed_auctions/closed_auction,
              $oi in $o/itemref/@item, $op in $o/price,
              $id in $i/@id
              satisfies ($oi=$id and data($op)<300))
            return <item>
               <iname>{N1.1.2.1}</><desc>{N1.1.2.2}</>
            </item>
N1.1.2.1:-  for $in in $i/name return $in
N1.1.2.2:-  for $id in $i/description return $id
```

Figure 6: XQ-Tree Representation t_1 for q_1

$depends_t(n) = ancestors_t(n)$. If $depends_t(n) = \phi$ for some class, there is no possibility that $q(n)$ has references to variables defined in other nodes.

For any XQ-Tree node n in t, $cq_t(n)$ denotes the *complete query* of n. $cq_t(n)$ is complete in the sense that it gives the complete computation of the XML node sequence corresponding to n. For example, $cq_{t_1}(\texttt{N1.1.1})$ returns elements for all category names. Before defining $cq_t(n)$, we define $compose(q, q')$ as follows: Given query fragments q and q', the `for (where)` clause of $compose(q, q')$ is the concatenation of `for (where)` clauses of q and q'. Then, $cq_t(n)$ is obtained by recursively applying function $compose(q, q')$ to the query fragments in $\{q(n)\} \cup \{q(m) | m \in depends_t(n)\}$. For example, $cq_{t_1}(\texttt{N1.1.1})$ = "`for $c in /site/ categories/category, $cn in $c/name return $cn`." In the following, given a query fragment $q(n)$ = "`for` $expr_f$ `where` $expr_w$ `return` v," we write $cq_t(n)$ as "`for` $expr'_f$, $expr_f$ `where` $expr'_w \wedge expr_w$ `return` v," where $expr'_f$ and $expr'_w$ are results of composing expressions taken from $\{q(m) | m \in depends_t(n)\}$.

Given a complete query q, we write $\{|q|\}$ to denote a *set* of XML nodes in q's evaluation result. $\{|cq_{t_1}(\texttt{N1.1.1})|\}$ is a set of XML elements for category names. The subscript t is omitted when there is no ambiguity.

4 General Framework

Taking into account that XQuery queries are complicated with various query constructs, one of our technical contributions is that we have developed a general framework that includes several concepts (XQ-Trees, contexts, extents) to make possible the nontrivial task of learning XQuery queries from examples. In this section, we explain the proposed framework.

The learning process has two phases. First, XLearner creates an *XQ-Tree skeleton* t_s from a template and dropped examples. t_s gives the tree structure of the XQ-Tree to learn. Then, it traverses t_s and learns query fragment $q(n)$ for each node n in t_s. Intuitively, each $q(n)$ in t_s is associated with one Drop Box in the template, and what

XLearner learns through membership and equivalence queries on a Data Box is a path expression and selection conditions for the associated $q(n)$.

4.1 Template and XQ-Tree skeleton

As explained in Section 2, XLearner creates a template based on a given target schema. The template works as a basis for an XQ-Tree skeleton. Figure 7 (a) is the template for our example. A template is a tree, in which a node is created for each element type in the target schema. Each node has an element type with it. For example, an element definition $A = (B, C, (D|E*))$ results in a node n_A with four children n_B , n_C, n_D and n_E.

Examples must be consistent with the schema; only one of D or E can take a dropped example in this case. In general, the template allows only one dropped example in each altenation structure, i.e., only one of the Drop Boxes in an alternation structure can exclusively accept a dropped example. If the user wants to show examples for different alternatives, there are two options:

Use the split operation. If the selection of alternatives depends on the choices of other dropped examples, use the `split` operation. For example, consider the case where a list of publications is the union of disjoint sets of journal papers and books. Each publication item in the list has a volume number in its child element if the publication is a journal paper, while it has a publisher's name if the publication is a book. In other words, the selection of child elements *depends on* other examples (journal papers or books). If the split operation is applied to a template, it splits a repetition structure into two consecutive copies of the structure (Figure 8). Then, the user can drop different examples into the copy of the alternation structure. The same discussion applies to the case of optional structure because $A = (B?)$ is the same as $A = (B|\varepsilon)$.

Use the context change operator. The operation is used when the new context is considered to be the same as the original context, in the sense that the context change does not affect the query. For example, you may not be able to find any item whose region is Africa, in the "book" category in the process of trying to show examples of African items. In this case, if there is an item whose region is Africa in "computer" category, the user can change the "book" context to "computer" by a mouse operation, and can use it as an example. Note that if there were an item whose region is Africa in the "Book" category, the user also wanted the item in the query result. In other words, the selection of examples in the alternative structure has no dependency on the context change.

If an element has recursive definitions, a new node is created for each new occurrence of the same element type. Because it is possible that the template's size is infinite, XLearner incrementally presents the template to the user. At first, only the first instance of recursively defined element is shown. When the user clicks on a node, it instantiates its children.

If the target schema guarantees that a node and its parent has one-to-one relationship, the edge connecting them is labeled as "1." The labels are used in the learning process. For simplicity, the following discussions assume that each node has at most one child connected by 1-labeled edge and there are no consecutive 1-labeled edges on

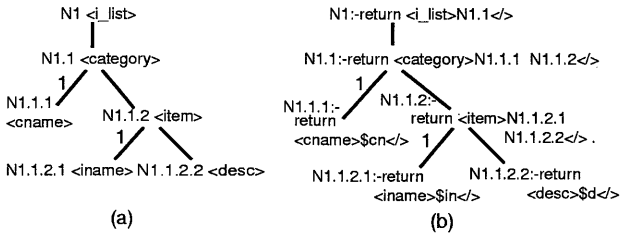


Figure 7: Template and XQ-Tree skeleton

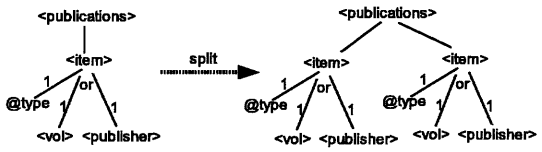


Figure 8: Split operation

any path from the root to a leaf. (The template in Figure 7 satisfies this restriction.) It is easy to remove the restriction.

An XQ-Tree skeleton is a simple XQ-Tree computed from the template and the dropped example nodes. Figure 7 (b) illustrates the XQ-Tree skeleton for the example in Section 2. The tree structure is the minimum subtree of the template including all the nodes into which examples were dropped. Note that while the size of the template can be infinite (if the target schema has recursive element definitions), every XQ-Tree skeleton is a tree with a finite number of nodes. For each XQ-Tree node n, $q(n) =$ "`return` $expr_r$." If an example node e is dropped into XQ-Tree node n, $expr_r$ contains v_e, which is a variable that corresponds to e. (In Figure 7(b), $v_{\texttt{book}}$ =\$cn, $v_{\texttt{H.Potter}}$ =\$in, and $v_{\texttt{BestSeller}}$ =\$d.)

4.2 Learning Each Query Fragment

We write n_e to denote an XQ-Tree node in the XQ-Tree skeleton t_s, where e is the example node dropped into n_e. XLearner learns the whole XQ-Tree by learning $q(n_e)$ for each n_e in turn. XLearner assumes that $q(n_e)$ has the form "`for` $expr_f$ `where` $expr_w$ `return` v_e." It is $expr_f$ and $expr_w$ that XLearner needs to learn for $q(n_e)$. XLearner knows in advance the class of queries to learn so that it can compute $depends(n_e)$ using t_s. We give formal definitions of the context and extent first.

Context. We define the context of example e as follows: Let an *assignment* be a set of pairs (v, o) where v is a variable and o is an XML node. Then, given an XQ-Tree skeleton t_s, a dropped example node e, and the other dropped nodes for t_s, we write $context_{t_s}(e)$ to be an assignment:

$$\{(v_{e_i}, e_i) | n_{e_i} \in depends_{t_s}(n_e)\}$$

where e_i is a dropped example node for XQ-Tree node n_{e_i}, and v_{e_i} is a variable that corresponds to e_i (in n_{e_i}'s `return` clause).

In the example scenario, $depends(n_{\text{H.Potter}})$ contains n_{book} (a node whose dropped node is "book"), and $context$("H. Potter") $= \{(v_{\text{book}}$, "book")$\}$. As mentioned, $depends(n_e)$ varies according to classes of queries, which will be explained in the following sections.

Extent. As explained in Section 2, $EXT_{e,context(e)}$ (abbreviated EXT_e) is a key concept of XLearner: XLearner tries to learn EXT_e and uses it to compute query fragment $q(n_e)$. Intuitively, EXT_e is a subset of $\{|cq(n_e)|\}$ that is characterized by $context(e)$. For example, consider $\{|cq_{t_1}(n_{\text{H.Potter}})|\}$ (i.e., $\{|cq_{t_1}(N1.1.2.1)|\}$ in Figure 6 where $cq_{t_1}(N1.1.2.1)$ is the composition of $q_{t_1}(N1)$, $q_{t_1}(N1.1)$, $q_{t_1}(N1.1.2)$, and $q_{t_1}(N1.1.2.1)$). While $\{|cq_{t_1}(n_{\text{H.Potter}})|\}$ is the set of names of *all* the items whose regions are `africa` or `europe` and that were sold for less than 300 dollars, $EXT_{\text{H.Potter},\{(v_{\text{book}},\text{book})\}}$ is the set of item names only in the context of category "book." In other words, $o \in EXT_e$ means not only $o \in \{|cq(n_e)|\}$ but o satisfies a certain relationship with the dropped example nodes in $context(e)$. In this example, $context_{t_1}$("H. Potter") $= \{(v_{\text{book}}$, "book")$\}$ is used.

The formal definition is as follows: Let $cq(n)$ = "`for` $expr'_f$, $expr_f$ `where` $expr'_w \wedge expr_w$ `return` v" (See Section 3) and $context(e) = \{(v_{e_1}, e_1), \ldots, (v_{e_m}, e_m)\}$. The extent $EXT_{e,context(e)}$ is then defined based on $cq(n_e)$ and $context(e)$ as:

$EXT_{e,context(e)} = \{|$ `for` $expr'_f$, $expr_f$ `where` $expr'_w \wedge expr_w \wedge (v_{e_1}$ `is` $e_1) \wedge \ldots \wedge (v_{e_m}$ `is` $e_m)$ `return` $v_e|\}$[2].

Note that in a special case where $depends(n_e) = \phi$, (and so $context(e) = \phi$), $EXT_{e,context(e)} = \{|q(n_e)|\}$.

Learning $q(n_e)$. The basic flow of learning each $q(n_e)$ is as follows: (1) XLearner makes membership queries to learn EXT_e. Based on the answers given so far, $\{q(n_{e_i})|n_{e_i} \in depends(n_e)\}$ and $context(e)$, it constructs a hypothesis:

$\hat{EXT}_{e,context(e)} = \{|$`for` $expr'_f, \hat{expr}_f$ `where` $expr'_w \wedge \hat{expr}_w$
$\wedge (v_{e_1}$ `is` $e_1), \ldots, (v_{e_m}$ `is` $e_m)$ `return` $v_e|\}$,

which is the same as EXT_e except that $\hat{expr}_f$ and $\hat{expr}_w$ are hypotheses of $expr_f$ and $expr_w$, respectively. (2) XLearner makes an equivalence query to ask if $\hat{EXT}_e = EXT_e$ (3) If $\hat{EXT}_e = EXT_e$, halt. XLearner can conclude $q(n_e)$="`for` $\hat{expr}_f$ `where` $\hat{expr}_w$ `return` v_e." Otherwise, go to (1). Note that in the step (1), $\hat{expr}_f$ and $\hat{expr}_w$ are modified based on counterexamples that claim $\hat{EXT}_e \neq EXT_e$.

The concepts of contexts and extents are introduced for learning $\hat{expr}_w$. Details are explained in Section 7.2.

[2] "v_1 `is` v_2" holds when v_1 and v_2 have the same identity.

5 Class X0 and its family

Given the general framework, challenging issues include how to traverse the XQ-Tree skeleton and how to combine known learning techniques for learning each query fragment. For explanation, we give an algorithm for a class of simple XQuery queries first and develop it for more general classes. This section gives algorithms for classes of XQuery queries without join/selection conditions.

X0, **X0***, **X0***+ are classes of XQ-Trees where $depends(n) = \phi$ for every XQ-Tree node n. No `where` condition is allowed in queries in the classes.

[X0] This is the simplest class of XQ-Trees with only one XQ-Tree node. An example XQ-Tree (and the corresponding query) in **X0** is as follows:

```
Query: for $i in /site/regions//item return <result>$i</>
XQ-Tree: N1:- for $i in /site/regions//item return $i
```

Before giving the **X0**'s definition, we define 0-$Learnable(n)$, a predicate about learnability of a query fragment. The predicate holds when $q(n)$ is learnable by a simple algorithm based on the Angluin's algorithm [2]. Formally, 0-$Learnable(n)$ holds iff $q(n)$ = "`for` v `in` p `return` v" where p is a regular path expression that starts with a `document` function (i.e., the root of an XML document). Note that 0-$Learnable(n_e)$ guarantees that $depends(n_e) = \phi$ and $EXT_e = \{|q(n_e)|\}$.

Definition An XQ-Tree t is in Class **X0** iff t consists of the root node n only and 0-$Learnable(n)$.

Algorithm LEARN-X0. In **X0**, $\hat{expr}_f$ ="v `in` $\hat{p}$." Therefore, $\hat{EXT}_e$={|`for` v_e `in` $\hat{p}$ `return` v_e|}, and $\hat{p}$ is a path regular expression, which is the only thing to learn to compute $\hat{EXT}_e$. Because learning regular expressions is equivalent to learning deterministic finite automata, it is natural to use a learning technique to learn a dfa.

We apply the Angluin's algorithm [2] to our context for efficiently learning a dfa from membership queries and equivalence queries. The algorithm first takes an example string, makes a number of membership queries on different strings, and constructs a hypothesis automaton $\hat{M}$ consistent with the strings. It makes an equivalence query on $\hat{M}$ to receive a counterexample (string) to modify $\hat{M}$, and repeats the process until it finds the intended automaton. The key idea is to attempt to continually discover new states. Since a given counterexample suggests that it leads to a wrong state in the current hypothesis automaton, the algorithm discovers a new state. Figure 9 shows a step of the learning process where a given positive counterexample (that should be accepted) is used to find a new state for a dfa.

In our context, a sequence of tags, which represents a path from the XML instance's root to an XML node, corresponds to a string in the Angluin's algorithm. Figure 9 gives hypothesis automatons generated in a process of learning a dfa that corresponds to /site/region/asia. The system first receives $path(e)$ for the dropped example e, where $path(e)$ is the sequence of *tags* that matches the sequence of XML nodes from the XML instance's root to e. It then makes membership queries and equivalence queries so that it can learn $\hat{p}$. Note that the algorithm makes membership queries and equivalence queries on a sequence s of tags, not an XML node. So XLearner has to choose an XML

element m s.t. $path(m) = s$ to issue an membership query. To make an equivalence query, XLearner highlights XML nodes in $E\hat{X}T_e$.
[**X0***] This includes XQ-Trees with more than one node.

Definition: An XQ-Tree t is in **X0*** iff for every node n in t, 0-$Learnable(n)$ holds.

Following is an example:

```
  for $i in /site/regions//item/
  return <result>$i
            for $c in /site/categories/category/name
            return <cname>$c</>
         </>

N1:-   for $i in /site/regions//item/
       return $i{N1.1}                          /* result */
N1.1:- for $c in /site/categories/category/name
       return $c                                 /* cname */
```

Every $q(n)$ has a variable in its `return` clause. Therefore, every node in an XQ-Tree skeleton requires a dropped example in the learning process. XQ-Trees in the class correspond to nested queries whose query results are essentially Cartesian products.
LEARN-X0*. **X0***'s definition says that for every node n_e, $depends(n_e) = \phi$ and $EXT_e = \{|q(n_e)|\}$. This implies that XLearner can traverse the XQ-Tree skeleton in an arbitrary order and learn each query fragment using **LEARN-X0**.
[X0*+] In the previous classes, every $q(n)$ is assumed to have a variable v_e in its `return` clause, because $q(n)$ is learned using the example e. We get **X0*+** when we relax the assumption and allow query fragments without variables under a certain condition. Following is an example:

```
for $c in /site/categories
return <root>
          <result> $c
             <name-list>
               for $n in /site//name return <name>$n</>
             </>
       </></>

N1:- for $c in /site/categories return {N1.1[1]} /* root */
N1.1:- return $c{N1.1.1}                        /* result */
N1.1.1:- return {N1.1.1.1}                   /* name-list */
N1.1.1.1:- for $n in /site//name return $n     /* name */
```

where "[1]" after a node identifier means the edge is labeled as 1 according to a given DTD. q(`N1`) and q(`N1.1.1`) have no variable in their `return` clause, but they are (indirectly) learnable because `N1` and `N1.1.1` satisfy a certain condition. Informally, such a node has to satisfy the following conditions (Figure 10): (1) If n has a child connected by 1-labeled edge (represented as $C^1(n)$), $q(n)$ is learnable by applying **LEARN-X0** to $collapse(n, C^1(n))$. Here, $collapse(n, n')$ be an XQ-Tree node whose query fragment, $q(collapse(n, n'))$, is defined by $compose(q(n), q(n'))$. This applies to `N1`, because 0-$Learnable(collapse($`N1,N1.1`$))$ holds. (2) Otherwise $q(n)$ is just a holder of child XQ-Tree nodes. This applies to `N1.1.1`.

Formally, we say 0-$Learnable'(n)$ holds when both of the following conditions hold:

(A1) $C^1(n)$ exists $\Rightarrow$ 0-$Learnable(collapse(n, C^1(n)))$

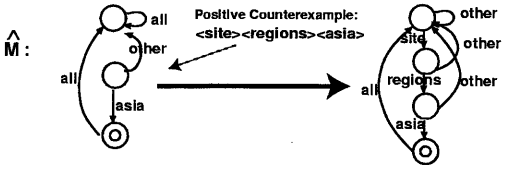


Figure 9: Finding a New State for a DFA

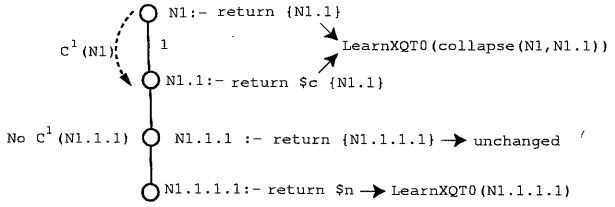


Figure 10: LEARN-X0*+

(A2) $C^1(n)$ does not exist $\Rightarrow$
$q(n) =$ `return` *node id's of n's children.*

Definition: An XQ-Tree t is in **X0*+** iff for every node n in t, 0-$Learnable(n) \vee$ 0-$Learnable'(n)$.

LEARN-X0*+. This is the same as **LEARN-X0*** except that it collapses nodes or returns an empty `return` clause when $q(n)$ in the XQ-Tree skeleton has no variable in the `return` clause. Figure 10 shows how to learn each query fragment when an XQ-Tree skeleton is given to learn the query example above.

The result of LEARN-X0*+ applied to the query example would be as follows:

```
N1':- for $c in /site/categories return $c{N1.1.1}
N1.1.1:- return {N1.1.1.1}
N1.1.1.1:- for $n in /site//name return $n
```

where `N1` and `N1.1` are collapsed into `N1'`. Note that XQuery's semantics guarantees that collapsing the nodes connected by 1-labeled edges does not change the query result.

6 Class X1 and its family

Class **X1** and its family deal with selection conditions in `where` clauses. We introduce the concept of *1-learnability* instead of 0-learnability and use it to define the classes.

Notations. When an XQ-Tree node has "`for` v `in` p," we write $expr(v)$ to denote the *binding expression* "v `in` p," and write $expr(v).path$ to denote the *path expression* p (e.g., $expr_{t_1}(\$cn).path$=`$c/name` in Figure 6). $Expr^*(v)$ is the closure of binding expressions to compute v. ($Expr^*_{t_1}(\$cn)$ = {"`$c in`

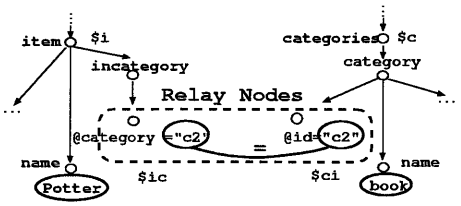


Figure 11: Relay Nodes

`/site/categories/category/`," "`$cn in $c/name`"}.) $expr^*(v)$ stands for a binding expression "v `in` p'," where p' is the concatenation of all path expressions appearing in $Expr^*(v)$. ($expr^*_{t_1}(\$cn)$ = "`$cn in /site/categories/category/name`.") $associated(v)$ is the set of variables appearing in $Expr^*(v)$. ($associated_{t_1}(\$cn) = \{\$cn, \$c\}$.) Let n_v be the XQ-Tree node whose `for` clause defines v. Then, $associatable(v)$ is the set of variables that appear in query fragments of $ancestors(n_v)$ or n_v. The XQuery's variable scope guarantees that (1) $associated(v) \subseteq associatable(v)$, and (2) if $v' \notin associatable(v)$, $q(n_v)$ cannot specify any relationship between v' and v.

1-Learnability. Intuitively, 1-$Learnable(n_e)$ holds when $q(n_e)$ has a particular form of conditions in a `where` clause. The property guarantees that $q(n_e)$ is learnable if $\{q(n)|n \in ancestors(n_e)\}$ and $context(e)$ are given. Formally, 1-$Learnable(n)$ holds iff $q(n)$ = "`for` v `in` p `where` $expr_w$ `return` v," where

(1) $expr^*(v).path$ is a regular path expression that starts from the document root (Note that p does not have to start from the document root), and

(2) $expr_w$ is $\bigwedge_{v' \in (associatable(v) - associated(v))} RS(\{v, v'\})$, where $RS(\{v_1, v_2\})$ is either $True$ or $RS'(\{v_1, v_2\})$. Here, $RS'(\{v_1, v_2\})$ stands for one of the following expressions that specifies a relationship between v_1 and v_2:

(Rel1) $data(v_1) = data(v_2)$

(Rel2) `some` w `in` v_1/q `satisfies` $RS'(\{w, v_2\})$

(Rel3) `some` w `in document()/`q `satisfies` $RS'(\{v_1, w\}) \wedge RS'(\{w, v_2\})$

where q is a path expression with the child axis and optional position numbers or `last()` function (e.g., `a[1]/b/c[last()]`).

The definition says that the condition in a `where` clause of $q(n_e)$ should be a conjunction of relationships between a variable v_e and $v' \in (associatable(v_e) - associated(v_e))$. It is also possible that two variables have an indirect association. **Rel2** and **Rel3** represent two patterns of indirect associations via w. Here, we refer to w as a *relay node* of the relationship $RS'(\{v_1, v_2\})$. Figure 11 illustrates indirect association between $\$c$ and $\$i$ through two relay nodes, where "`some $ic in $i/incategory/@category satisfies (some $ci in $c/@id satisfies ($ic=$ci))`" holds.

The point here is that, given an XML instance (e.g., Figure 11) and a pair of XML nodes ("H. Potter" and "book") in it, we can enumerate *all* the predicates that can

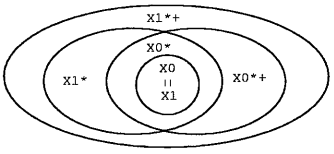


Figure 12: Relationship among Classes

appear in the `where` clause of $q(n)$ if we know that 1-$Learnable(n)$ holds. Note that 1-$Learnability$'s definition implies that (1) only equality predicates and simple path expressions are contained in the `where` clause, both of which can be derived from the given XML instance in a straightforward way, and that (2) the number of possible predicates is limited.

Note that 0-$Learnable(n)$ is a special case where $expr^*(v).path = p$ and there is no condition in the `where` clause, so 0-$Learnable(n) \Rightarrow$ 1-$Learnable(n)$. In addition, if n is the root, $depends(n) = \phi$ and $expr_w = True$. Therefore, 1-$Learnable(n) \wedge Root(n) \Rightarrow$ 0-$Learnable(n)$.

[X1] An XQ-Tree t is in Class **X1** iff t consists of the root node n only and 1-$Learnable(n)$. Since 1-$Learnable(n) \wedge Root(n) \Rightarrow$ 0-$Learnable(n)$, **X1=X0**.

[X1*] An XQ-Tree t is in **X1*** iff for every node n in t, 1-$Learnable(n)$.

[X1*+] An XQ-Tree t is in **X1*+** iff for every node n in t, 1-$Learnable(n) \vee$ 1-$Learnable'(n)$, where 1-$Learnable'(n)$ is defined in a similar way to 0-$Learnable'(n)$.

The relationship among the classes is shown in Figure 12. The XQ-Tree in Figure 6 (ignoring the part in the box) is in **X1*+**.

7 LEARN-X1*+

1-Learnability's definition and the XQuery's semantics say that $q(n)$ may refer to variables defined in $ancestors(n)$. Therefore, $depends(n) = ancestors(n)$ when 1-$Learnable(n)$ holds. This suggests that algorithms to learn queries have to take care of *traversal order* of the XQ-Tree.

LEARN-X1*+ works in the same way as **LEARN-X0*+** except that (1) it traverses the given XQ-Tree skeleton in the depth-first order and (2) it incorporates **LEARN-X1** instead of **LEARN-X0** to learn the query fragment $q(n)$ for each XQ-Tree node n. **LEARN-X1** learns $q(n)$ when 1-$Learnable(n)$ holds. It differs from **LEARN-X0** in that (1) it takes as input $context(n)$ and (2) its outputs (query fragments) contain `where` clauses.

In the following subsections, we explain main components of the **LEARN-X1*+** algorithm. Detailed discussions on **LEARN-X1*+** are given in Appendix B.

7.1 LEARN-X1

To keep the discussion simple, we represent each query fragment in an equivalent normal form. Remember that the algorithm **LEARN-X0** relies on the fact that $expr_f.path$ of a query fragment "`for` $expr_f$ ..." is restricted to be a path expression *starting from*

the document root. In **X1**'s family, it is no longer true. To make it possible to take a similar approach, we replace each $expr_f$ with $expr^*(v)$, which is guaranteed to start from the document root by the 1-$Learnable(n)$'s definition. More precisely, we use a slight extension of $expr^*(v)$ to specify *path-sharing constraints* among variables. For example, we represent q_{t_1}(N1.1.2.1) (in Figure 6) as follows:

```
for $in in /site{$i1}/regions{$i2}/
                  (europe|africa){$i3}/item{$i4}/name
where {$i4 is $i}
return $in
```

where the "/site{$i1}/regions{$i2}/(europe | africa){$i3}/item {$i4}/iname" is an abbreviation of "$i1 in /site, $i2 in $i1/regions, $i3 in $i2/(europe | africa), $i4 in $i3/item, $in in $i4 /iname." Note that $in is defined by $expr^*(v)$ that starts from the document root, and that the *path-sharing constraint* between $in and $i that they share the same item element, is specified by the predicate $i4 is $i. In contrast, in the original notation (in Figure 6), the path-sharing constraint is specified by the fact that the path expression of binding expression "$in in $i/name" starts with $i. Therefore, we always assume that each path expression in $expr_f$ starts from the document root. How to obtain path expressions in the normal form is discussed in Appendix A.

We use the following notational convention: n_e is the XQ-Tree node whose query fragment ($q(n_e)$) is to be learned. e is the example object that has been dropped to n_e. v_e is the variable that corresponds to e (which is assigned in a return clause in the XQ-Tree skeleton). $context(e) = \{(v_{e_1}, e_1), \ldots, (v_{e_m}, e_m)\}$. $expr_f =$ "v_e in p" and $expr_w = c$. Then, $q(n_e) =$ "for v_e in p where c return v_e," and EXT_e = {|for $expr'_f$, v_e in p where $expr'_r \wedge c \wedge (v_{e_1}$ is $e_1) \wedge \ldots \wedge (v_{e_m}$ is $e_m)$ return v_e|}.

We represent the condition c in the where clause of $q(n_e)$ as a *set* of predicates, and interpret c as the conjunction of the predicates in c.

Note that if $c \supseteq c'$, it is guaranteed that c is *stronger* than c' in the sense that {|for v in p where c return v|} is contained in {|for v in p where c' return v|}.

Overview. Figure 13 illustrates the data flow among modules to implement **LEARN-X1**. The user has EXT_e in mind (Figure 13(a)). The algorithm learns EXT_e by receiving answers to (1) membership queries about EXT_e and (2) equivalence queries about the relationship between EXT_e and hypothesis $\hat{EXT}_e$. In order to compute $\hat{EXT}_e$ (Figure 13(b)), the algorithm must compute $\hat{p}$ and $\hat{c}$. P-Learner computes $\hat{p}$. C-Learner (Section 7.2) takes as input $context(e)$ and the *Interaction History Table* (IHT_e) to compute $\hat{c}$.

Figure 14 gives more detail on the relationship among **LEARN-X1** and related modules. Here, it takes as input n_e to compute $q(n_e) =$ for v_e in $\hat{p}$ where $\hat{c}$ return v_e. P-Learner implements **LEARN-X0** to compute $\hat{p}$ and records interactions with the user in IHT_e for learning EXT_e. (An abstract algorithm of P-Learner is given in Figure 15. Details of Angluin's algorithm are omitted there.) C-Learner computes $\hat{c}$. IHT_e is used to compute $\hat{p}$ and $\hat{c}$. The basic flow is as follows: First, an XQ-Tree node n and the dropped example node e (for EXT_e) is given to the function **LEARN-X1**(n_e) in Figure 15. The function passes "($tags(path(e))$, positive)" to P-Learner (line 2, see Section 5). P-Learner raises MQs and EQs to compute $\hat{p}$.

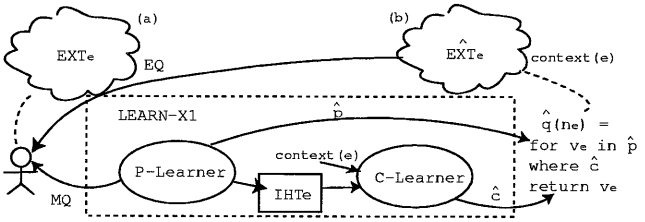


Figure 13: Data Flow in Learning $q(n_e)$

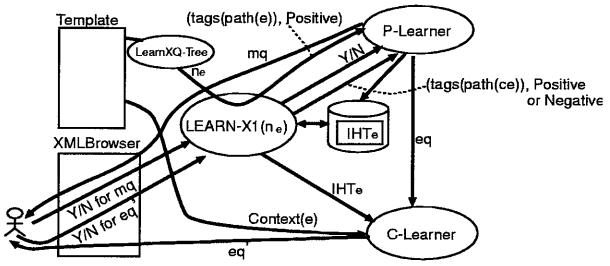


Figure 14: Relationship among LEARN-X1(n_e) and related modules

Each MQ mq is passed to the user via the XML browser to ask if some XML node m is in EXT_e. The answers (Y or N) are returned to P-Learner (line 14. see Figure 14 also). P-Learner records the answers in IHT_e (line 15-16). How to process the answers is explained later. EQs are not passed directly to the XML Browser, because an EQ on EXT_e requires condition $\hat{c}$ to compute $E\hat{X}T_e$. (P-Learner merely computes eq(="`for` v_e `in` $\hat{p}$ `return` v_e") (line 2)) Therefore, eq is first passed to C-Learner (line 3), which is responsible to compute $\hat{c}$. C-Learner makes EQ eq' by adding "`where` $\hat{c}$" to eq. Here, $\hat{c}$ is computed using IHT_e and $context(e)$. The counterexample to eq' is recorded in IHT_e by **LEARN-X1**(n_e) and then passed to either P-Learner or C-Learner, according to an algorithm (Figure 17) explained later. Those steps are repeated until the user accepts an EQ.

Interaction History Table. $IHT_e(Node, Ans, P, C)$ is a relation that records the user's answers to learner's queries for EXT_e. Each tuple t in IHT_e records one answer (interaction) to a learner's query. Attribute $Node$ stores an XML node. Attribute Ans stores the answer to a learner's query. For example, if Y is given to XLearner as an answer to an membership query on node m, then $t.Node = m$ and $t.Ans = Y$. It also records the reasons for the answer. If $t.Ans = N$ because the path expression p does not accept $t.Node$, $t.P = N$. If $t.Ans = N$ because $t.Node$ does not satisfy the condition c, $t.C = N$. If $t.Ans = Y$, both $t.P$ and $t.C$ must be Y. IHT_e always includes (e, Y, Y, Y) to record the fact that the dropped example node e is a positive example of EXT_e.

A Mismatch Problem. Given EXT_e, we write $PATH_e$ to be $\{|$`for` v_e `in` p `return` $v_e|\}$, where p is learned by interactions about EXT_e. Note that $PATH_e = EXT_e$ for queries in **X0** and its family. Therefore, XLearner can directly get an-

```
1. LEARN-X1(n_e) {
2.   eq=P-Learner(tags(path(e)),positive);
3.   eq'=C-Learner(eq, IHT, context(e));
4.   ans = equivalenceQuery(eq');
5.   while (ans!=OK) {
6.     eq = processCounterExAndRecord(ans);
7.     eq'= C-Learner(eq, IHT, context(e));
8.     ans = equivalenceQuery(eq');
9.  }
10. }
11.
12. P-Learner(s: sequence of tags,
              flag: Positive or Negative) {
13.   for each membership query mq
                          on a sequence s of tags {
14.     ans = membershipQuery(m)
                    where m in {m|tags(path(m))=s};
15.     if (ans==Y) insert (o,Y,Y,Y) into IHT;
16.           else insert (o,N,N, null) into IHT; // ans==N
17.   }
18.   eq= newEquivalenceQuery();
19.   return eq;
20. }
21.
22. C-Learner(eq, IHT, context) {
23.   eq'= eq with a where clause having
                                  the strongest condition
        consistent with select{Ans=Y}(IHT) and context;
24.   return eq'
25. }
```

Figure 15: Abstract Algorithm to Learn $q(n_e)$

swers about $PATH_e$ when the user gives answers about EXT_e. But in the case of **X1** and its family, there is a mismatch between $PATH_e$ and EXT_e (Figure 16 (a)). It always holds that $EXT_e \subseteq PATH_e$, because EXT_e is associated with additional conditions to select XML nodes. The mismatch problem makes naive application of **LEARN-X0** impossible. Consider the case where P-Learner makes an MQ to learn the intended path expression p. P-Learner selects some sequence s of tags, select some node m in the XML instance s.t. $tags(path(m)) = s$, and asks the user whether $m \in PATH_e$, although the user assumes the question is whether $m \in EXT_e$. It is okay if the answer is Y, because $m \in EXT_e \Rightarrow m \in PATH_e$. But if the answer is N, there are two possibilities: (**Case M#1**) $m \notin PATH_e$, (**Case M#2**) $m \in PATH_e$ but $c \wedge (v_{e_1}$ `is` $e_1) \wedge \ldots \wedge (v_{e_m}$ `is` $e_m)$ is not satisfied. The case **M#2** occurs for the "Encyclopedia" explained in Section 2. When XLearner asks if "Encyclopedia" node is included in $EXT_{\texttt{Potter}}$, the user would give N, because he wants items whose sales price was less than 300 dollars. Therefore, although "Encyclopedia" is in $PATH_{\texttt{Potter}}$, it is not in $EXT_{\texttt{Potter}}$.

Note that it is impossible to distinguish between the two cases at the time the answer N is given. Therefore, the algorithm first assumes the case is **M#1** (in other words, it assumes $PATH_e = EXT_e$ at first), and when it finds an inconsistency in IHT_e, it backtracks and corrects the tuple. The tuple $t = (m, N, N, null)$ is always inserted when the answer is N, where $t.C = null$ means that the interaction implies nothing on c.

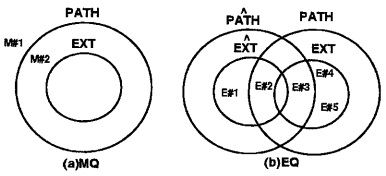


Figure 16: Mismatch between $PATH$ and EXT

P-Learner copes with the mismatch by recording given answers into IHT_e and checking its consistency when receiving answers to EQs.

Equivalence Query (EQ) ($E\hat{X}T_e = EXT_e$?)**.** Making an EQ requires $\hat{c}$ as well as $\hat{p}$ to compute $E\hat{X}T_e = \{|$ `for` v_e `in` $\hat{p}$ `where` $\hat{c}$ `return` $v_e|\}$, because XLearner needs to highlight XML nodes in $\{|EXT_e|\}$ in an EQ to ask the user if $E\hat{X}T_e = EXT_e$ (see Section 2). Again, P-Learner requires answers about whether $P\hat{ATH}_e = PATH_e$ to compute $\hat{p}$, but the user gives answers to inform whether $E\hat{X}T_e = EXT_e$.

If $E\hat{X}T_e = EXT_e$ the learning algorithm stops with success. Otherwise, it receives a counterexample ce. Due to the mismatch between $PATH_e$ and EXT_e, the story is a little complicated; Figure 16 (b) illustrates all the 5 cases that can happen when a counterexample ce is given. In the following explanations, IHT_e^c represents the snapshot of IHT_e when ce is given.

(1) When ce is a negative counterexample:

Case E#1: $ce \notin PATH_e$. Because this implies that $P\hat{ATH}_e \neq PATH_e$, ce serves as a counterexample for $P\hat{ATH}_e$. Note that the case implies nothing on the condition $\hat{c}$, because there may be XML nodes that satisfy $\hat{c}$ even outside $PATH_e$. Therefore, $(ce, N, N, null)$ has to be inserted into IHT_e.

Case E#2: $ce \in PATH_e$. This implies that the condition $\hat{c}$ lacks some condition to exclude ce from EXT_e, and that ce cannot serve as a counterexample for $P\hat{ATH}_e$. Therefore, (ce, N, Y, N) has to be inserted into IHT_e.

Note that the algorithm actually cannot distinguish **E#1** and **E#2** when ce is given, because it is on the way to learn (unknown) $PATH_e$ and cannot directly check if $ce \in PATH_e$. However, if XQ-Trees are in **X1*+**, we do not encounter **E#2**, because C-Learner is designed to output $\hat{c}$ as the *strongest conditions* that is permitted in **X1*+**. (We explain it in Section 7.2.) Therefore, the algorithm always insert $(ce, N, N, null)$ into IHT_e. However, we have to take care of the case **E#2** when we extend the expressive power, which will be explained in Section 9.

(2) When ce is a positive counterexample: Always (ce, Y, Y, Y) has to be inserted to IHT_e, because $ce \in EXT_e$ implies $ce \in PATH_e$ and the condition in the `where` clause is satisfied.

Case E#3: $ce \in P\hat{ATH}_e$. (Note that the algorithm can identify this case when ce is given, in contrast to Cases **E#1** and **E#2**.) The case implies that the condition $\hat{c}$ is too strong, but does not imply $P\hat{ATH}_e$ is wrong. The algorithm calls only C-Learner to make a new (weaker) condition $\hat{c}'$ that is consistent with all examples including ce.

Case E#4: $ce \notin P\hat{ATH}_e \wedge \neg\exists t \in IHT_e^c(tags(path(ce)) = tags(path(t.Node)) \wedge t.P = N)$. This implies that $P\hat{ATH}_e \neq PATH_e$. The ce can be used as a positive counterexample to recompute $P\hat{ATH}_e$.

```
processCounterExAndRecord(ce) {
 if (ce is negative) {
   insert (ce, N, N, null) into IHT;
   return P-Learner(tags(path(ce)), Negative);
 else { // ce is positive
   insert (ce, Y, Y, Y) into IHT;
   switch (Case) {
     E#3: return C-Learner(eq, IHT,context);
     E#4: return P-Learner(tags(path(ce)), Positive);
     E#5: find the 1st tuple $t=(m,N,N,null) in IHT
                    s.t. tags(path(ce))=tags(path(m))
          replace $t with (m,N,Y,N);
          exit to backtrack to
                 the state just after $t is inserted.
   }
 }
}
```

Figure 17: Algorithm to Process Counterexample ce

Case E#5: This is the case where $ce \in PATH_e$ but there exists $t_{wrong} \in IHT_e$ s.t. $(tags(path(ce)) = tags(path(t_{wrong}.Node)) \wedge t_{wrong}.P = N)$. This means that the user have stated before that $t_{wrong}.P = N$. This may look strange because the answers are inconsistent about p, but can happen because P-Learner inserts $(m, N, N, null)$ into IHT_e even in case **M#2**, instead of (m, N, Y, N). When the algorithm encounters **E#5**, it replaces t_{wrong} with (m, N, Y, N), backtracks to the state and restarts from there. Note that some of tuples in IHT_e inserted after t_{wrong} can be reused in the second execution; The algorithm avoids asking the user for answers to learner's queries when it has answers in those tuples.

Figure 17 summarizes the algorithm to process a counterexample ce against equivalence query eq'.

7.2 C-Learner

C-Learner outputs the strongest condition $\hat{c}$ consistent with all positive example nodes stored in IHT_e. Because condition $\hat{c}$ is a conjunction of predicates, we can map $\hat{c}$ into a monotone (no negation) k-term $x_1 \wedge \ldots \wedge x_k$, where each predicate in $\hat{c}$ corresponds to a variable in the term. There is an algorithm to learn a monotone k-term that makes equivalence queries at most k times (Figure 18). In the algorithm, if i is included in A, x_i must be true. The algorithm always makes an equivalence query with the strongest condition so that every counterexample (an assignment to $y_1, \ldots, y_k$) is guaranteed to be a positive counterexample. Note that a counterexample can remove a number of unnecessary variables at one time.

We apply the algorithm to our context, using predicates instead of variables. The first issue to consider is how to derive the set of candidate predicates that can be included in $\hat{c}$. Let an assignment a_0 be $context(e) \cup \{(v_e, e)\}$. Intuitively, a_0 represents a situation where e is accepted as a member of EXT_e. C-Learner computes the set of candidate predicates by computing the set of *all* predicates that hold about relationships between the dropped example e and all the XML nodes in $context(e)$. Predicates used in 1-$Learnable(n)$ are all considered. We use $cond(context(e), (v_e, e))$

```
A = {1,...,k}
while (true) {
   Let ĉ = ∧_{i∈A} x_i.
   Make an equivalence query with ĉ.
   If the answer of the query is ''OK'' (i.e., ĉ = c) {
      halt
   } else {
      let y_1,...,y_k be the counterexample.
      A = A ∩ {i|y_i = true}
   }
}
```

Figure 18: Algorithm to Learn Monotone k-term from Positive Counterexamples

to denote the set of candidate predicates. (How to compute the set is explained later.) Note that $cond(context(e), (v_e, e))$ is the "strongest" set of predicates for the first hypothesis (i.e., $k = |cond(context(e), (v_e, e))|$ in the algorithm). Therefore, $cond(context(e), (v_e, e))$ may have unnecessary predicates that happen to hold only in a_0. When this occurs, C-Learner eliminates such predicates using counterexamples.

The second issue is what we should use as counterexamples in Figure 18 to compute $\hat{c}$. As mentioned, C-Learner uses $context(e)$ and IHT_e to compute $\hat{c}$. Each assignment $a_i \in \{context(e) \cup \{(v_e, t.Node)\} | t \in \sigma_{Ans=Y}(IHT_e)\}$ represents a situation where a positive example node for EXT_e is accepted. We use the set of predicates that hold in each situation as a counterexample. Formally, for each $t \in \sigma_{Ans=Y}(IHT_e)$, $cond(context(e), (v_e, t.Node))$ serves as a counterexample. In the setting, the algorithm outputs the strongest condition consistent with all situations where the positive examples are chosen.

Computing $cond(context(e), (v_e, e_i))$**.** The computation is done with the aid of a *data graph*. It is a graph structure of XML similar to the XQuery and XPath data model [19], where a tree structure represents an XML instance. In addition, the data graph has edges between nodes having the same value. We call such an edge a *v-equality edge*. Figure 11 is an example fragment of a data graph. C-Learner enumerates all the predicates that hold about relationships among example nodes. Note that conditions in `where` clauses are restricted to be a particular form by 1-*Learnable(n)*'s definition, and the number of possible predicates is limited.

The main concern about the algorithm is management of the data graph. First, keeping all of the v-equality edges among nodes requires a large amount of additional data. Second, enumerating all paths between example nodes has an exponential complexity.

We can use heuristics and known techniques to cope with the issues.

Keeping v-equality edges. In practice, combinations of values used as join conditions are limited. In relational databases, it is typical that indices are prepared for such values. So it is natural that we assume there are indices (such as Vindex [14]) for such values in XML repositories. In XMark and XML Use Cases, 3 out of 27 equi-joins are based on explicit idref-id references. The rest are based on element contents, but the contents are something that are intended to serve as foreign keys.

Computing paths between nodes. In typical XQuery queries, the length of paths between example nodes is not long. The maximum length in the Xmark and XML Use Cases is 12. So we need to traverse only 6 edges from each examples. In addition,

we can use a data structure similar to Graph Schemas [6] to restrict the search space. Graph Schemas are graphs to describe partial knowledge of a graph structure. If there is no path between different element types in the graph schema, there is no path at the instance level either.

8 Reducing the Number of Interactions

We denote by p, n and k a regular expression, the number of states in the minimal dfa representing p, and the number of characters over which the language is defined. In XLearner, k corresponds to the number of XML element types. Let m be the length of the longest counterexample received. In our context, the length of a counterexample ce is the length of a tag sequence from the root to ce. For the Angluin's algorithm, the number of learner's queries is in $O(kmn^2)$. This means learning a simple regular path expression (corresponding to a dfa with a small number of states) could require hundreds of learner's queries. The polynomial number of interactions is not sufficient for XLearner, because it is a real-world interactive system.

XLearner reduces the number of the learner's queries by using properties specific to our context. Specifically, it uses the following two rules to automatically give default answers for the learner's queries without asking the user for answers. As explained in Section 10, the rules dramatically reduce the number of learner queries.

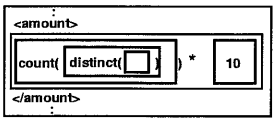
R1: If Angluin's algorithm makes an membership query on sequence s of tags and there is no m in the XML instance s.t. $path(m) = s$, N (No) is given to the algorithm. The current prototype uses the Relax NG for filtering, but other forms of metadata such as Graph Schema can be used as well.

R2: If a positive example m has been given to P-Learner, and the last tag of $path(m)$ is $t1$, N is given to the algorithm as answers to membership queries on s whose last tag is not $t1$. If XLearner receives another positive counterexample with the last tag $t2(\neq t1)$ for an equivalence query, XLearner backtracks and uses a new assumption that the last tag matches any kind of node. Finally, if XLearner receives a negative counterexample under the new assumption, XLearner discards all the assumptions and gives no more default answers. This is based on a heuristic that the last component of a typical path expression is likely to be a tag.

9 Expressive Power and an Extension

This section relaxes the assumption that the user is a minimally adequate teacher, which may be too strong for practical use. We extend **LEARN-X1*+** by allowing it to receive the following three types of explicit specifications.

(1) Functions in Drop Boxes. So far, a Drop Box is assumed to be a place where the user drops an XML node taken from the XML Browser. In general, however, mapping of an XML schema to another sometimes involves functions, such as aggregation functions. We now introduce the concept of a *Nested Drop Box* and allow the user to explicitly specify functions. If the user types the name of functions in a Drop Box, XLearner opens a new (nested) Drop Box corresponding to each parameter of the func-




```
Nx:- for ...
     return count(Nx.1)*10
Nx.1:- for $v in distinct(Nx.1.1)
       return $v
Nx.1.1:- for $w in ... return $w
```

Figure 19: Nested Drop Box

return value\parameter	atomic	collection
atomic	inc	sum
collection	–	distinct,union

Figure 20: Classification of functions

tion (Figure 19 (left)). According to the type of return values and parameters of a given function, XLearner rewrites the structure of the XQ-Tree (Figure 19 (right)), allows the user to drop example nodes in the Drop Boxes, and learns the query fragments from simple yes-or-no interactions.

Let n be an XQ-Tree node and $f(x_1, \ldots, x_n)$ be the function given to the Drop Box corresponding to $q(n)$'s `return` clause. Funcions can be classified according to the types of their return values and parameters. Figure 20 shows how some of functions are classified. The rules for rewriting the XQ-Tree are as follows: (1) If the return value of $f(x_1, \ldots, x_n)$ is a set (or sequence) value, the query fragment $q(n)$ becomes `for $v in` $f(x_1, \ldots, x_n)$ `return $v`. The `distinct function` in Figure 19 is an example. Otherwise, the function is kept in the `return` clause (`*10` function in Figure 19). (2) If a parameter x_i is a set (or sequence) value, a new XQ-Tree node is created as a child of the current node (For the `count` function in Figure 19, the node `Nx.1` is created. For the `distinct` function, the node `Nx.1.1` is created). Otherwise, the parameter remains in the original position. XLearner recursively applies the rules and the same learning process is applied to learn query fragments for the new nodes.

(2) Keys for Sorting. We allow XLearner to receive keys for sorting through a dialog box called an *OrderBy Box*. An OrderBy Box appears when the user clicks on an element to be sorted. After receiving the sort key(s), XLearner inserts an `order by` clause into the query fragment.

(3) Explicit Selection/Join Conditions. This is specified in a *Condition Box*, as explained in Section 2. Note that in contrast to the other types of explicit specifications, XLearner automatically learns a certain kind of join condition. The question, then, is how to integrate the learning schema and the Condition Box.

Interestingly, XLearner can determine when a Condition Box should be used. Remember that the algorithm does not encounter the case **E#2** as long as queries are in **X1*+**. Now the story is different. Because we allow any form of condition other than those allowed in **X1*+**, it is no more guaranteed that C-Learner constructs the strongest conditions, and we have to consider **E#2**. Remember that XLearner cannot distinguish **E#2** from **E#1** when a counterexample is given. Therefore, it always assumes that the case is **E#1** as it does for the case **M#1** or **M#2** (see Section 7.1). If the assumption

is wrong, XLearner will eventually encounter the case **E#5** and find an incorrect tuple in IHT_e. The procedure to handle the case **E#5** can cope with the inconsistency, no matter whether the incorrect tuple is due to the wrong assumption of **E#1** or **M#1**[3].

The extended algorithm displays a Condition Box when it finds **E#2**. The reason is that C-Learner is designed to output the strongest condition that is expressible in **X1*+**, so there should be some other conditions beyond those given by C-Learner in the case **E#2**.

There are two types of Condition Boxes: a Positive Condition Box (PCB) and a Negative Condition Box (NCB). A PCB is used to specify why the dropped positive example is included in the extent, while an NCB is used to specify why the negative counter example is *not* included in the extent. The user can choose either type. The one used in Section 2 is a PCB. If the user specifies a condition c with an NCB, condition $\neg c$ is used as a condition. NCB is especially useful when the user wants to use `empty` predicate[4] in a condition, because there is no positive example node that can be used as its parameter. The subexpression in the box in Figure 6 can be created by this extension.

With this extension of the basic framework, XLearner has a practical expressive power. Let **X1*+E** be a class of XQ-trees learnable by **LEARN-X1*+** with the extension. Then, we define a set $\mathbf{XQ_I}$ of XQuery queries as follows: Given an XML instance **I**, an XQuery query Q is in $\mathbf{XQ_I}$ if there exists a query Q' s.t. $Q'(\mathbf{I}) = Q(\mathbf{I})$ and the XQ-tree representation of Q' is in **X1*+E**. The example query in Figure 2 (including the box part) is in $\mathbf{XQ_I}$. Note that $\mathbf{XQ_I}$ is parameterized by instance **I**; it is natural for the set to depend on **I** because we discuss a system that receives *real* examples from an XML instance.

Figure 21 shows the percentages of queries in XMark and XML Use Cases that are included in $\mathbf{XQ_I}$. Specifically, they include 19 out of the 20 XMark queries[5], and 11 out of 12 XML Query Use Case "XMP" (Experiences and Exemplars) queries. The main reason for 0% in Use Case "NS" is that the queries contain special matching patterns that use namespaces. The queries in Use Case "PARTS" contain recursive user-defined functions. The queries in Use Case "STRONG" exploit information on strongly typed data.

Name	Percentage
XMark	95% (19/20)
UC "XMP"	91.7%(11/12)
UC "TREE"	83.3%(5/6)
UC "SEC"	60%(3/5)
UC "R"	77.8%(14/18)
UC "SGML"	100%(11/11)
UC "STRING"	50%(2/4)
UC "NS"	0%(0/8)
UC "PARTS"	0%(0/1)
UC "STRONG"	0%(0/12)

Figure 21: Expressive Power of XLearner

[3] Actually IHT_e records which case produced each tuple.

[4] The `empty` predicate returns true if the parameter is an empty sequence.

[5] For Q18 that uses a user-defined function, XLearner learned an equivalent, but different query without any user-defined function.

10 Experiments

We counted the number of interactions required for learning queries and evaluated the effect of the mechanisms to reduce the number. We show the results for the 19 queries in XMark (Figure 22(top)) and the 11 queries in XML Query Use Case "XMP" (Figure 22(bottom)). The experimental results for other queries were similar and omitted. We chose those sets of queries because they contain *a variety of typical query fragments*.

XMark

	D&D(#t)	MQ	CE	CB(#t)	OB	Reduced(R1,R2,Both)
Q1	1(1)	5	1	1(3)	0	2434(2412,486,464)
Q2	1(1)	0	1	1(4)	0	2439(2416,486,463)
Q3	2(2)	0	1	1(13)	0	4878(4832,972,926)
Q4	1(1)	0	1	1(9)	0	1627(1608, 405,386)
Q5	1(2)	0	1	1(3)	0	1627(1612,405,390)
Q7	3(8)	10	0	0	0	7449(7382,1458,1391)
Q8	2(3)	0	0[1]	0	0	2604(2573,729,698)
Q9	2(2)	0	0[2]	0	0	4051(4023,881,853)
Q10	12(12)	0	0[3]	0	0	26994(26756,5589,5351)
Q11	2(3)	0	1	1(5)	0	4066(4025,891,850)
Q12	2(3)	0	2	2(8)	0	4066(4025,891,850)
Q13	2(2)	10	0	0	0	4868(4822,972,926)
Q14	1(1)	5	1[2]	1(3)	0	2426(2404,486,464)
Q15	1(1)	3	0	0	0	12637(12604,1053,1020)
Q16	1(1)	1	1	1(2)	0	2438(2422,486,470)
Q17	1(1)	0	1	1(2)	0	1177(1161,405,389)
Q18	1(2)	0	0	0	0	1627(1608,405,386)
Q19	2(2)	10	0	0	1	4848(4804,972,928)
Q20	4(8)	0	4	4(14)	0	6508(6420,1620,1532)

XML Query Use Case "XMP"

	D&D(#t)	MQ	CE	CB(#t)	OB	Reduced (R1,R2,Both)
Q1	2(2)	0	1	1(3)	0	250(236,80,66)
Q2	2(2)	0	0	0	0	250(234,80,64)
Q3	2(2)	0	0	0	0	250(234,80,64)
Q4	2(3)	0	1	1(3)	0	250(234,80,64)
Q5	3(3)	0	1	1(3)	0	356(334,112,90)
Q7	2(2)	0	1	1(3)	1	250(236,80,66)
Q8	2(2)	0	1	1(3)	0	250(234,80,64)
Q9	1(1)	2	1[3]	1(3)	0	26(23,8,5)
Q10	2(5)	0	0	0	0	106(98,32,24)
Q11	4(4)	0	2	2(6)	0	106(98,32,24)
Q12	2(2)	0	1	1(10)	2	126(112,60,46)

Figure 22: The Number of Interactions for Learning

In Figure 22, **D&D** is the number of dropped example nodes. Note that the user is allowed to specify an arbitrary function in each Drop Box. Therefore, we need a measure of its complexity. We use the number of terminal nodes in the function's abstract syntax tree. Terminal nodes include function names, values, and dropped example nodes. For example, the number of terminal nodes in $multiply(plus(30, 40), 2)$ equals 5. #t in the parenthesis is the number of terminal nodes. **MQ** is the number of membership queries, and **CE** is the number of counterexamples given by the user. **CB** is the number of Condition Boxes that are invoked to specify explicit selection/join conditions. We give the numbers of terminal nodes again. **OB** is the number of required `OrderBy` boxes. **Reduced** is the number of interactions reduced by **R1** and **R2**

introduced in Section 8. Numbers of interactions that can be reduced by either rule is given as **both**. These measurements depend on static and dynamic factors. By static, we mean the factors independent of the interactions; they are the number of element types defined in the DTD (**MQ** is affected. See Section 8.), the size of XQ-tree (all measurements), the number of states in the minimum dfa corresponding to each path regular expression (**MQ**, **CE**), the density of the data graph (affecting the number of predicates enumerated by C-Learner, thus **CE**), and the regularity of the data structure (**Reduced**). Note that the size of the data graph is *not* included in the factors. Dynamic factors mean which XML nodes the user gives as counterexamples against equivalence queries (**MQ**, **CE**, **Reduced**) and which XML nodes to choose for membership queries (**CE**). Measurements shown in the figure are basically the "best-case" measurements with the XML instances. If the "worst-case" measurement is worse than the "best-case," the "worst-case" is shown in square brackets[6].

Following is a key observation: Rules **R1** and **R2** dramatically reduce the number of interactions, thereby showing that our approach can be practical. **R1** suppresses membership queries if they are inconsistent with the XML instance. Therefore, the more regular the structure of the XML instance, the fewer membership queries XLearner will ask. The structures of data in XMark and XML Use Cases are relatively regular, so the rule eliminates a great number of interactions. The regularity also accounts for the reason why there is no difference between the best and worst cases in **MQ** and **Reduced**, which are affected by the structure of paths from the document root to examples.

Another important observation is that, in practice, it does not matter so much which example is chosen as a counterexample. The reason is that the data graph is relatively sparse and the performance of C-Learner is little affected by the choice of a counterexample. This also explains why **CE** is always 0 or a small number and is not a function of **D&D** (the size of the XQ-Tree skeleton) in the results. Theoretically, **MQ** scales linearly with the number of **D&D**. But because experimental results show that XLearner requires 0 to only a few interactions for learning each typical query fragment, we can say that XLearner requires a relatively small number of interactions for a general XQuery query, even if it is larger than the queries used in the experiments. For example, Figure 22 shows that learning Q10 consisting of 12 XQ-Tree nodes requires at most three interactions after the user drags and drops example nodes.

11 Discussion and Conclusion

XLearner is more appropriate for *restructuring* than *retrieval*. Using real examples is very intuitive; we developed a Web-site construction tool in a similar approach and demonstrated that even users with no knowledge of database-related concepts could specify fairly complex data manipulations [13]. The user, however, must always find examples. We believe the proposed approach is complementary to other retrieval-oriented approaches and we can combine other approaches and XLearner in appropriate ways. One possible extension is to incorporate known search mechanisms into

[6]We examined the XML instances and selected appropriate example nodes by hand for those cases.

XLearner to find examples that satisfy given conditions. Another direction is to allow the user to specify "artificial" examples in the spirit of QBE. Details of such extensions are beyond the scope of this paper.

In this paper, we explained XLearner, a tool for developing XML mapping queries using machine learning techniques. Specifically, we presented algorithms for learning several classes of XQuery queries. We also proposed some extensions and showed that our most elaborate algorithm has a practical expressive power and can learn a large set of XQuery queries. Beyond that, we presented experimental results where the number of required interactions is small. Future work includes development of a mechanism to reuse past interactive operations.

Acknowledgments

We would like to thank Akiyoshi Nakamizo for participating in the implementation. The research has been supported in part by the Grant-in-Aid for Scientific Research from JSPS and MEXT.

References

[1] T. Amoth, P. Cull, P. Tadepalli. Exact Learning of Unordered Tree Patterns from Queries. *Proc. Computational Learning Theory*, pp. 323-332, 1999.

[2] D. Angluin. Learning regular sets from queries and counterexamples. Information and Computation, 75(2):87-106, 1987

[3] D. Angluin. Computational Learning Theory: Survey and Selected Bibliography. *Proc. 24th Annual ACM Symposium on Theory of Computing*, pp. 351-369, 1992.

[4] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with Data Values: Typechecking Revisited. *Proc. PODS*, pp. 138-149, 2001.

[5] H. Arimura, H. Sakamoto, S. Arikawa. Efficient Learning of Semi-structured Data from Queries. *Proc. Algorithmic Learning Theory*, pp. 315–331, 2001.

[6] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. *Proc. ICDT*, pp.336-350, 1997.

[7] A. Blum, S. Rudich. Fast Learning of k-Term DNF Formulas with Queries. *Proc. ACM Symp. on Theory of Computing*, pp. 382-389, 1992.

[8] A. Doan, P. Domingos, A. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. *Proc. SIGMOD*, pp. 509-520, 2001.

[9] E.M. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37, pp.302-320, 1978.

[10] H. Ishizaka. Learning simple deterministic languages. *Proc. Computational Learning Theory*, pp. 162-174, 1989.

[11] R. Kosala, J. Van den Bussche, M. Bruynooghe and H. Blockeel. Information Extraction in Structured Documents using Tree Automata Induction. *Proc. Principles of Data Mining and Knowledge Discovery*, 2002.

[12] M. Sipser. Introduction to the Theory of Computation. PWS Publishing Company, 1997.

[13] A. Morishima, S. Koizumi, H. Kitagawa and S. Takano. Enabling End-users to Construct Data-intensive Web-sites from XML Repositories: An Example-based Approach. *Proc. VLDB*, pp. 703-704, 2001.

[14] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajamaran. Indexing semistructured data. Technical report, Stanford University, Computer Science Department, 1998.

[15] Online Computer Library Center. Introduction to Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/ about_the_ddc.htm`

[16] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, R. Fagin. Translating Web Data. VLDB 2002: 598-609

[17] Yasubumi Sakakibara. Learning Context-Free Grammars from Structural Data in Polynomial Time. Theoretical Computer Science, 76(2-3), pp. 223-242, 1990.

[18] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, R. Busse. Why And How To Benchmark XML Databases. SIGMOD Record 30(3): 27-32 (2001)

[19] W3C. XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/query-datamodel/.

[20] W3C. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases.

[21] W3C. XML Path Language (XPath). http://www.w3.org/TR/xpath.

[22] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/ TR/xquery/.

[23] W3C. XML Query Requirements. http://www.w3.org/TR/xmlquery-req.

A Translating DFA's into path expressions

XLearner translates DFA's into regular expressions in order to generate path expressions. Since the internal form of path expressions has to incorporate path-sharing constraints as explained in Section 7.1, it is required that outer-most operators of the obtained regular expressions are concatenations. In other words, given a DFA G, we need a regular expression in the form of $e = e_1 \cdot e_2 \ldots e_n$ where $\cdot$ is the concatenation operator and G recognizes $L(e)$. For that purpose, we preprocess the given DFA before applying the standard algorithms [12] to our problem.

In the preprocessing phase, we decompose the given G into a set of dfa's $G_1, G_2, \ldots G_n$ as shown in Figure 23 where $L(G) = L(G_1) \cdot L(G_2) \cdot \ldots \cdot L(G_n)$. Then, e can be obtained as $e = re(G_1) \cdot re(G_2) \cdot \ldots \cdot re(G_n)$ in the next phase. Here, $re(G_i)$ is a regular expression corresponding to G_i.

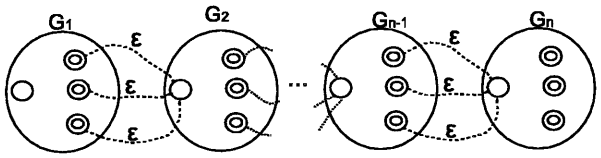


Figure 23: Preprocessing Result

We explain how this can be done. Let's consider the conditions with which we can decompose a DFA G into G_1 and G_2 (Figure 24), where $L(G) = L(G_1) \cdot L(G_2)$. In other words, the language recognized by G is the Cartesian product of those recognized by G_1 and G_2. The following holds: Let s_a, s_b, and s_c be the set of states in G s.t. every paths from the starting state to the accepting states goes through one of the states. Let G_a (or G_b, G_c) be the minimum subgraph of G that has all the paths from s_a (or s_b, s_c) to the accepting states (Figure 25 (a)). Then, when $L(G_a) = L(G_b) = L(G_c)$ we can divide G. The s_a, s_b, and s_c become the accepting states of G_1. One of G_a, G_b, and G_c becomes G_2. Note that in our settings, G has the minimum number of states. Therefore, it holds that $G_a - s_a$, $G_b - s_b$, and $G_c - s_c$ are identical and that the transitions starting from each of s_a, s_b, and s_c to states in $G_a (= G_b = G_c)$ are completely the same in terms of the labels and destinations (Figure 25 (b)). The discussion applies to the cases where we have any number of s_i's.

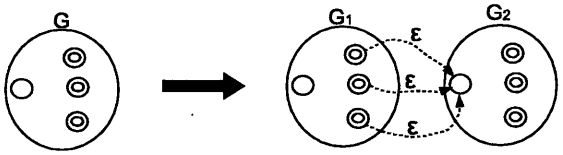


Figure 24: Dividing G into G_1 and G_2

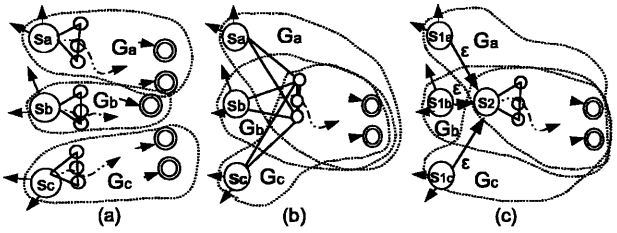


Figure 25: States in G

There are two cases: (1) Destinations of all of the transitions starting from each of s_a, s_b and s_c are states in $G_a - s_a$. Since the given DFA has the minimum number

of states, it holds that $s_a = s_b = s_c$ and $G_a = G_b = G_c$. An important point is that every path from the starting state to the accepting states should go through the state $s_a (= s_b = s_c)$. (2) Some of the transitions starting from each of s_a, s_b, and s_c are the same but the others are not. In this case, we can devide s_a (or s_b, s_c) into $s1_a$ (or $s1_b$, $s1_c$) and $s2$ like Figure 25 (c). Here, every path from the starting state to the accepting states goes through the state s_2.

Based on that observation, XLearner divides G using as clues the states that every path from the starting state of G to accepting states goes through. Figure 26 is an example of the preprocessing phase. First, XLearner identifies and checks off the states that every path from the starting state of G to accepting states goes through. Next, it decomposes G according to that information.

XLearner uses the decomposition result to construct a regular path expression. The result for the example in Figure 26 is $(abc) * \cdot a \cdot (\epsilon|b) \cdot d \cdot e$.

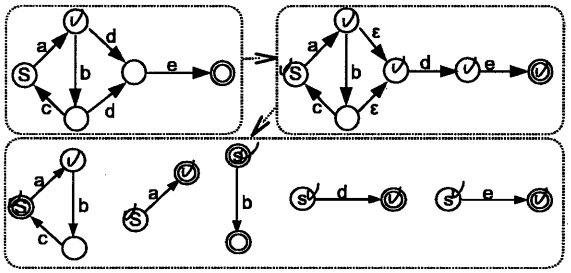


Figure 26: DFA Decomposition Example

B Detailed Discussions on LEARN-X1*+

This appendix gives detailed discussions on how **LEARN-X1*+** learns any query $t \in$ **X1*+**. That is, for any interactions consistent with query $t \in$ **X1*+**, **LEARN-X1*+** outputs $t' \in$ **X1*+** s.t. $t'(\mathbf{I}) = t(\mathbf{I})$.

Theorem. LEARN-X1*+ learns any query $t \in$ **X1*+**.

We prove the theorem by induction on the XQ-Tree's structure.

Basis: If n is the root of the XQ-tree t, LEARN-XQT1*+ learns $q(n)$, because $depends(n) = \phi$ and the algorithm simply uses the Angluin's algorithm to learn regular path expressions.

Induction step: If n is not the root and $q(parent(n))$ is learned, $depends(n) \neq \phi$ and the algorithm's traversal order guarantees that we know $\{q(m) | m \in depends(n)\}$. Given the assumptions, **LEARN-X1*+** learns $q(n) =$ `for` v `in` p `where` c `return` v, if the two following conditions hold:

1. P-Learner gets appropriate answers to learn p in each case (M#1, M#2, E#1, E#3, E#4, E#5). In other words, it is guaranteed that P-Learner can get appropriate answers to membership and equivalence queries. The cases M#1, E#1 and E#4 cause no problem since counter examples can be directly used as inputs to P-Learner. Therefore, we have to guarantee the following two conditions hold.

(a) If the algorithm makes a wrong assumption that M#2 is M#1, the algorithm eventually encounters the case E#5 to correct it, and

(b) E#3 does not prevent P-Learner receiving appropriate answers later.

2. C-Learner outputs correct conditions, which is true if:

(a) C-Learner can enumerate all possible predicates included in the strongest condition, and

(b) if $\hat{c}$ is stronger than c, $EXT - \hat{EXT} \neq \phi$ (trivial).

If all the statements are proved to be true, **LEARN-X1*+** learns all query fragments in t, and the proof is complete.

Proof of 1(a). Consider the case where the algorithm inserted $t = (m, N, N, null)$ into IHT for $s = tags(path(m))$, although it should have inserted (m, N, Y, N) (i.e., it was the case **M#2**). $\hat{p}$ is constructed to reject s, based on the answers recorded in IHT. However, p should accept s, because $t = (m, N, Y, N)$ was the correct tuple. If there exists an XML node m' in the instance **I** s.t. $s = tags(path(m'))$ and $t' = (m', Y, Y, Y)$, $\hat{p}$ rejects s and m' is not included in $\hat{EXT}$, although m' should be included in EXT. Therefore, m' serves as a positive counterexample that causes case **E#5**. If there exists no such m' in **I**, it is ok that $\hat{p}$ remains different from p, since $q(n)(\mathbf{I}) = \hat{q}(n)(\mathbf{I})$.

Proof of 1(b). LEARN-X1*+ does not give a counterexample ce to P-Learner if ce causes **E#3**. We prove here that giving such a counterexample does not prevent P-Learner from taking appropriate counterexamples. There are three cases when the algorithm encounters **E#3**.

Case 1: $\hat{p} = p$ (on I). The only possible case that the system will encounter after this **E#3** is **E#3**. The algorithm will eventually receive "ok" after taking counterexamples causing **E#3**.

Case 2: There exists a counterexample that works as a negative counterexample for $\hat{PATH}$. That is, $(\hat{PATH} - PATH) \cap (\hat{EXT} - EXT) \neq \phi$ on **I**. When the algorithm encounters **E#3**, it computes a *weaker* condition $\hat{c'}$ by removing some predicates from $\hat{c}$. As as result, the new extent $\hat{EXT'}$ satisfies the condition $\hat{EXT'} \supset \hat{EXT}$, and $(\hat{PATH} - PATH) \cap (\hat{EXT'} - EXT) \neq \phi$ on **I**. Therefore, there remains a counterexample that works as a negative counterexample against $\hat{PATH}$.

Case 3: There exists a counterexample that works as a positive counterexample for $\hat{PATH}$. That is, $(PATH - \hat{PATH}) \cap (EXT - \hat{EXT}) \neq \phi$ on **I**. Always $\hat{EXT} \subseteq \hat{PATH}$ by definition, and in the worst case, the algorithm makes $\hat{c'}$ so that $\hat{EXT'} = \hat{PATH}$ in handling **E#3**. Therefore, let's consider the worst case where $\hat{EXT'} = \hat{PATH}$. Let A be $(PATH - \hat{PATH}) \cap (EXT - \hat{EXT})$ and $A \neq \phi$. Then, $n \in \hat{PATH} \Rightarrow n \notin A$, and $(PATH - \hat{PATH}) \cap (EXT - \hat{EXT'}) = (PATH - \hat{PATH}) \cap (EXT - \hat{PATH}) = A \neq \phi$. Therefore, there remains a counterexample that works as a positive counterexample for $\hat{PATH}$.

Proof of 2(a). First, algorithm **LEARN-X1*+** outputs every variable that can be included queries in **X1*+**. Second, the algorithm enumerates all possible predicates that can be included in $RS(\{v, v'\})$ for the following reason: The 1-Learnablity's definition says that $RS(\{v, v'\})$ is defined only when v is the variable that appears in a `return` clause and $v' \in associatable(v) - associated(v)$. Because the algorithm guarantees that for every w appearing in a query $\exists u(w \in associated(u) \wedge u$ appears in a `return` clause), it holds that for all v' s.t. $v' \in associatable(v) - associated(v)$, $\exists u(v' \in associated(u) \wedge u$ appears in a `return` clause $\wedge\ u \neq v)$. This implies that there are always positive examples (including dropped examples) corresponding to v and $u(\neq v)$ and v' is bound to an XML node on a path between v and u. Therefore, predicates in $RS(\{v, v'\})$ are those that hold on a path between two different positive example nodes, which $cond(context(e), o)$ can enumerate.