EquiX—Easy Querying in XML Databases

Sara Cohen* Yaron Kanza* Yakov Kogan* Werner Nutt†
Yehoshua Sagiv* Alexander Serebrenik*

1 Introduction

The Web is explored by many users, but only few of them have experience in using query languages. Thus, one of the greatest challenges, provided by XML, is to create a query language simple enough for the naive user. In this paper, we present EquiX—a powerful and easy to use query language for XML. The main goal in designing EquiX is to strike the right balance between expressive power and simplicity.

EquiX has a form-based GUI that is constructed automatically from the DTD of XML documents. Query forms are built from well known HTML primitives. The result of a query in EquiX is a collection of XML documents, and it is automatically generated from the query without explicit specification of the format of the result. Knowledge of XML syntax is not required in order to use EquiX. Yet, EquiX is able to express rather complicated queries, containing quantification, negation and aggregation.

We present the data model in Section 2. In Sections 3, and 4 we introduce the query language syntax, and semantics. Generation of query results is discussed in Section 5. In Section 6, we conclude.

2 Data Model

We define a data model for querying XML documents [BPSM98]. We view a DTD as a scheme describing a set of XML documents conforming to it. Thus, we expect all XML documents that conform to a given DTD to have a somewhat similar structure.

We assume that each DTD has a designated element, called the *root element* of the DTD. We say that an XML document, x, *matches* a DTD, d, if x conforms to d, and the root element of d is the outermost element of x. We assume, w.l.o.g. that each DTD has a root element and it is named "root." Given a DTD specifying elements $e_1, \ldots, e_n$, we can create a DTD with the root element "root" by adding $<!$ ELEMENT root $(e_1|e_2|\ldots|e_n)* >$. We give an example of a DTD with a root element, describing a university document below. We will use this DTD in the sequel.

```
<!ELEMENT root       (department+)>
<!ELEMENT department (name, course*)>
<!ELEMENT course     (name, teacher+)>
<!ATTLIST course     course_id ID #REQUIRED>
<!ELEMENT teacher    (name)>
<!ELEMENT name       (#PCDATA)>
```

2.1 XML Database

In this section, we define the notion of a database of XML documents. It contains lists of XML documents matching the given DTDs. Intuitively, this is similar to the relational model, where a database contains lists of tuples conforming to the given relation schemes. We formalize this intuition below.

Definition 2.1 (Catalog) *A* catalog *is a triple, $< d, n, X >$, where d is a DTD, n is a unique identifier of d, called the* name *of d and X is a set of XML documents, each of which matches d.*

A *database* is a set of catalogs. This data model is natural and has useful characteristics. We have assumed that each XML document conforms to at least one DTD. This is a common assumption

*Institute for Computer Science, The Hebrew University, Jerusalem 91904, Israel.
†German Research Center for Artificial Intelligence GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany

(see [PV99]) and, it implies that the data is of a partially known structure. This knowledge will help in efficiently computing queries. In addition, since the basic structure of documents is known, we can display this information for the benefit of the user. Thus, the task of retrieving the information within the database does not require a preliminary step of querying the database for discovering its structure.

We have assumed that the database contains only specified documents. Thus, we do not need to traverse links within documents in order to find additional documents. This justifies the use of a closed-world assumption, i.e., information that we have not found is not stored in the database. When using a link traversal process, this assumption may not be justified, since there is no a priori bound on the number of links that have to be traversed in order to find any given data item. A closed world assumption is important for computing negation and universal quantification.

An XML document, when parsed, can be viewed as a tree[1]. We formalize this below. We first present an auxiliary definition of a labeled rooted tree. We assume $\mathcal{L}$ is a set of *labels*.

Definition 2.2 (Labeled Rooted Tree) *A* labeled rooted tree *is a tuple* $G = (N, E, r, l)$*, where*

- N *is a set of nodes;*
- $E \subseteq N \times N$ *is a set of directed edges that form a tree;*
- r *is the* root *of the graph, i.e., every node in* N *is reachable from* r*;*
- $l : N \rightarrow \mathcal{L}$ *is a labeling function that associates a label in* $\mathcal{L}$ *with each non-terminal node.*

Definition 2.3 (Document Tree) *Let* d *be an XML document and let* $G = (N, E, r, l)$ *be a labeled rooted tree. Suppose that* α *maps terminal nodes to atoms and to attribute values appearing as data in* d*, and* l *maps non-terminal nodes to element and attribute names appearing in* d*. Then* $D = (G, \alpha)$ *is the* document tree *of* d *if* G *is the parse tree of* d *and* α *associates terminal nodes of* G *with the data they represent in* d*.*

3 Query Syntax

We present the syntax of EquiX. EquiX is based on query forms generated by HTML. An EquiX query form is created by interactively exploring given DTDs. The user fills in text fields to specify constraints in a QBE-like style. In addition, the user specifies quantification of elements in the query and chooses elements to appear in the output. Intuitively, the user creates an "example" of the document that he is searching for, and EquiX finds all the documents that match this example. In Figure 1, we present an EquiX query, built from the DTD presented above, which computes the courses not taught by Dr. Jekyll.

3.1 Exploring DTDs

The user chooses a catalog from the database that holds the documents that she is interested in querying. The DTD of the catalog is explored to create a "template" of the document that is being searched for. In the beginning, the "root" element is displayed. The user opens a given element by clicking it, and EquiX displays the attributes and elements defined within the given element. Indentation is used to show the nesting of elements. Some elements may be opened in several ways. For example, an element may be defined as a choice between several elements or an optional element. In such cases, EquiX allows the user to decide how the element should be opened. If, in a DTD, an element is defined as appearing any number of times, the user can choose how many times the element should appear in the query (the button "add" shown in Figure 1 is used for that purpose). Attributes or elements defined as PCDATA, called *terminal elements*, appear on the screen with a text field. This allows the user to enter constraints on the data. Note that in Figure 1 the DTD is partially explored (i.e., some elements are opened).

After exploring the DTD, the output is defined by choosing elements to be projected. Projected elements are marked by adding the printing sign, *Pr*, at the left of the names of the elements selected for projection. The result document is automatically created by the query evaluation process.

[1] Although a document may contain ID and IDREF attributes, we do not treat them differently from CDATA attributes. Thus, a document is a tree and not a graph. This is natural since a DTD is considered a scheme and document connections through ID and IDREF values are not apparent in a DTD.

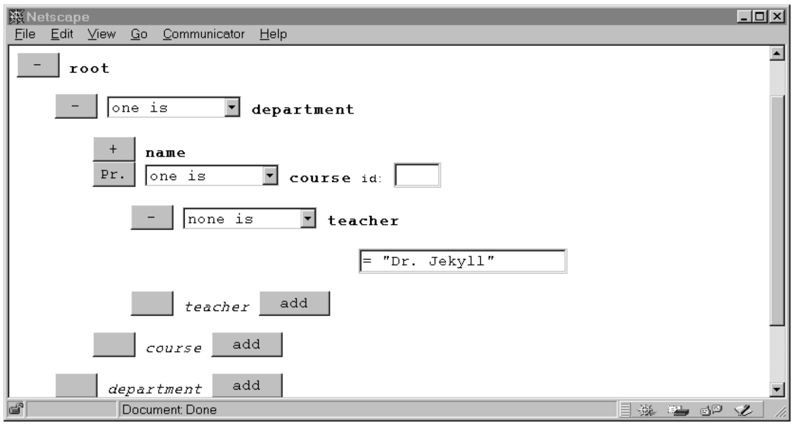

Figure 1: EquiX Query

3.2 Constraining Attributes and Terminal Elements

Attributes and elements defined as PCDATA are constrained in text fields. Text fields may contain conditions, variable definitions, and aggregate functions. A variable is denoted by the $ sign, followed by a sequence of alphanumeric characters, e.g., $\$x$. A constant is a word, a quoted phrase, or an alphanumeric prefix followed by the wild card symbol $*$. The wild card stands for any alphanumeric suffix.

An atomic condition is θt, where t is either a variable or a constant and θ is one of $\{<, >, \leq, \geq, =, \neq\}$. A condition is an atomic condition or a boolean expression (i.e., conjunction, disjunction, negation) of conditions. Comparisons are interpreted as numerical comparisons, if t is a number (or is bound to a number), and as comparisons with respect to lexicographical ordering, if t is a string. In addition, we interpret equality as substring matching. Note that there are no known types in XML documents. Thus, if a numerical comparison is performed against string data, the result will always be false.

A user may define a variable in a text field. Variables are bound to values that match the fields of the query form in which those variables are defined (see Section 4). Thus, each variable used in the query must be defined exactly once. A variable, $\$x$, is defined by typing *def* $\$x$.

We allow the following aggregation functions in a query: *count*, *sum*, *max*, *min*, and *avg*. To apply an aggregation function to an attribute or a terminal element, the user simply types the function name in the corresponding text field. The argument of the function is the element in which the function is defined. EquiX ignores data that violate the type constraint imposed by the aggregation function.

3.3 Constraining Non-Terminal Elements

Non-terminal elements (more precisely the edges leading into those elements) are quantified. In addition they may also be aggregated with the *count* function. There are four quantifiers in EquiX, namely, *One Is*, *None Is*, *All Are*, *Not All Are*—a user friendly way of expressing $\exists$, $\neg\exists$, $\forall$ and $\neg\forall$. Quantification is specified in a selection box located at the left of element names. The *count* function is activated similarly. Note that we only allow the *count* function on elements. All other aggregation functions are applied to atomic values, and non-terminal elements do not have atomic data associated with them.

4 Query Semantics

We translate query forms to trees in order to facilitate query processing. A query may be created by using several forms. To simplify our presentation, we assume that each query is created by one form. Thus, we define a *query tree* and not a query forest, which would be necessary if several forms were used.

Definition 4.1 (Query Tree) *A* query tree *is a tuple,* $T = (G, cond, quant, agg, P)$*, such that* $G = (N, E, r, l)$ *is a labeled rooted tree, where:*

- *cond is a partial mapping of terminal nodes to conditions and variable definitions;*
- *quant maps each edge to a quantifier in* $\{\exists, \forall, \neg\exists, \neg\forall\}$*;*
- *agg is a partial mapping of edges to aggregation functions in {count, sum, avg, max, min};*
- $P \subseteq N$ *is the set of projected nodes of the query.*

A query form is translated to a query tree in the obvious manner. Thus, the structure of the form (i.e., the hierarchy of elements and attributes) imply the structure of the query tree. The conditions in the form are represented by the function *cond*, and the projected elements of the form are represented by P. Quantification and aggregation of elements are represented by the functions *quant* and *agg*, respectively. Edges leading into elements are mapped by the functions *quant* and *agg* to quantifiers and aggregation functions, respectively.

We consider the similarities between query trees and SQL queries. Query forms are derived from the catalogs of a database. Thus, the documents in the chosen catalog represent the FROM clause. We can view *cond* as representing the WHERE clause of the query, since it imposes conditions on the values of XML documents. If *quant* maps all edges to $\exists$, then the query tree represents a SELECT-FROM-WHERE query. But when other quantifiers are used, one would have to use nested queries in order to express the same query in SQL. The usage of quantifiers in EquiX query forms makes it much easier to express complex queries compared to SQL. In SQL, projected attributes and aggregation are expressed in the SELECT clause. In our formalism, aggregation is represented by the function *agg*, and the nodes that are included in the result are represented by P. We assume that for all nodes, n, if $n \in P$, then the parent of n is also contained in P. Clearly, P can easily be extended if this is not the case.

Recall that a query is created by exploring a DTD appearing in the database. This DTD has XML documents associated with it in the database. We describe the computation of a query on a single document. This process should be repeated for each document associated with the DTD.

For a given document, query processing can be viewed as a process of singling out the nodes of the document tree that will be part of the output. Thus, given a query and a document tree, the procedure `Compute_Query`, which is described below, computes a set P of the nodes of the document tree that will be included in the result. The result is a document tree, that can be viewed as a subtree of the document.

Definition 4.2 (Result Tree) *Let* $D = (G, \alpha)$ *be a document tree, where* N *is the set of nodes of* G*. Suppose that the procedure* `Compute_Query` *is applied to* D *and returns* $P \subseteq N$*. Then the* result tree *is* $R = (G, \alpha, P)$*.*

For simplicity, we first assume that the query contains neither variables nor aggregation. For this special case, we define the procedure that creates the result tree from the document tree. Given a query tree $Q = ((N_q, E_q, r_q, l_q), cond, quant, agg, P_q)$ and a document $D = ((N_d, E_d, r_d, l_d), \alpha)$, the result of computing Q on D is the result tree $R = ((N_d, E_d, r_d, l_d), \alpha, P_d)$, such that P_d is defined by running `Compute_Query(`r_q`,`r_d`)` as described in Figure 2.

As a post-processing step, we consider nodes in P_d that have no sons in P_d. In this case, we add all of their sons to P_d. This is done since data values appear in the leaves and, hence, it is rather meaningless to want to see an interior nodes without the leaves below that node.

The procedure Compute_Query runs in a top-down fashion. There is also a bottom-up algorithm that runs in polynomial time, even when the query is considered as part of the input. This may seem surprising, since it might seem that quantifiers would make the problem more difficult. Roughly speaking, however, query evaluation in this case is analogous to evaluating a first-order query with only unary predicates, even though the evaluation is applied to a document tree that represents connections between elements.

Compute_Query can be generalized to deal with variables and aggregation. Note that in the general case, query evaluation is polynomial only if we assume that the query is fixed (i.e., data complexity). Adding aggregation alone does not effect the complexity of the computation. Thus, a query without variables and with aggregation can still be computed in polynomial time. Note that when computing a query with aggregation, the grouping variables (i.e., elements) must be defined. To simplify this process for the user, EquiX groups according to the lowest element, above the aggregation function, that was projected by the user. This is a natural choice for grouping, since it takes advantage of the tree structure of the document, and thus suggests a polynomial algorithm. In the result of a query with aggregation, an edge with the aggregation value is added to the grouping element.

```
boolean
Compute_Query(n_q, n_d):
if n_q is a terminal node
        if (cond(n_d) and n_q ∈ P_q) then P_d := P_d ∪ {n_d};
        return cond(n_d);
else
        let S be the set of sons of n_q in Q;
        for each s ∈ S do:
             let T_s be the set of sons, t, of n_d in D such that l_d(t) = l_q(s);
             switch quant((n_q, s)):
                  ∃:  status_s := ⋁_{t∈T_s} Compute_Query(s,t);
                  ∀:  status_s := ⋀_{t∈T_s} Compute_Query(s,t);
                  ¬∃:  status_s := ¬(⋁_{t∈T_s} Compute_Query(s,t));
                  ¬∀:  status_s := ¬(⋀_{t∈T_s} Compute_Query(s,t));
        if (⋀_{s∈S} status_s)
             if (n_q ∈ P_q) then P_d := P_d ∪ {n_d};
             return true;
        else
             remove all descendants of n_d from P_d;
             return false;
```

Figure 2: Query Processing Algorithm

5 Creating Query Results

The result of an EquiX query is a catalog, i.e., a triple containing a DTD, a name and XML documents. In this section, we define the XML documents and the DTDs that are the result of query processing. We do not require the user to perform restructuring in computing the result. Thus, querying is made easier.

When applying a query to an XML document, a result tree is created and it has the form (G, α, P), where P is the set of nodes in the original document tree that should appear in the result. Thus, the resulting XML document is simply the projection of the original document tree on the nodes of P. This projection is performed while retaining the ordering of the original document with respect to the elements projected out. Queries are applied to several documents, and thus a set of documents is created. Note that when aggregation functions are applied, the aggregation values are also added to the created documents according to their location in the result tree.

We now consider the creation of the DTD for the resulting catalog, called the *result DTD*. This DTD is based on definitions in the DTD from which the query was created, called the *original DTD*. As a first step, the elements in the query that are ancestors of projected elements, are also marked as projected.

The result DTD contains definitions for a subset of the elements appearing in the original DTD. For each element that will appear in the result DTD, we define its content. Defining an element's content is done in stages. Hence, for each element, e, that will appear in the result DTD, we create several *partial content definitions*, $d_1, \ldots, d_n$. In the result DTD, e is defined as <!ELEMENT e $(d_1|\ldots|d_n)$>.

We create a partial content definition for each projected element, e, by taking the sequence of the names of the projected elements directly below e. It may be the case that an element, e', below e, originated from a definition containing $e'*$ or $e'+$. In this case, the partial content definition of e contains $e'*$ or $e'+$, respectively. We now consider the projected elements that are not ancestors of other projected elements. EquiX includes in the result all the data in the document that is below such elements. For such elements, the partial content definitions created were empty. We define their partial content definitions as their content definitions in the original DTD. We continue this process recursively with the elements occurring in the partial content definitions that were just added.

The attributes appearing in the result DTD are those in the original DTD, but all attributes are defined as #IMPLIED. For elements for which an aggregation value is defined in the result tree, we add an attribute with this value. We add the aggregate value as an attribute, since the aggregation value can be seen as a property of the grouping element.

6 Conclusion and Related Work

In defining EquiX, we created an easy to use query language. We defined a form-based language, since in our experience, filling forms is the easiest interface for naive users. In addition, EquiX automatically creates result documents from queries, without user intervention. Moreover, EquiX also creates automatically a DTD for the result documents. Since specifying the structure of the result is not an easy task, this is another advantage of EquiX for naive users. EquiX is a simple, yet powerful query language. EquiX allows aggregation, quantification and negation. These can be used extensively in a query. EquiX creates results that fit its data model. Thus, iterative querying is possible (cf. [GW98]).

Most XML query languages proposed thus far (e.g., XML-QL [DFF+98], XQL [RLS98], XQL [IKK98]) do not have graphical interfaces. They have complex grammars, and thus are not suitable for naive users. In addition, they require either a priori knowledge of the DTDs or querying of the metadata. This adds an additional source of difficulty in the querying process. XML-GL [CCD+98] has a graph-based query language. However, graphs are not well known entities for non-scientists. In XMAS [BLP+98, LPVV99], querying can be done either graphically (in a form-based language) or textually. However, even in its graphical query language, the format of the result must be specified. In addition, aggregation and quantification are not easily formulated in the graphical interface of this language, and may even require a textual formulation. Thus, EquiX is noteworthy both in its added simplicity and in its querying abilities.

A natural question is to define the exact expressive power of EquiX. Another problem is to find a necessary and sufficient condition for tractable query processing. EquiX does not find minimal DTDs for its results. This is another area for future research. We are in the process of implementing EquiX.

References

[BLP+98] C. Baru, B. Ludäscher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xmas.html`.

[BPSM98] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, 1998. Available at `http://www.w3.org/TR/1998/REC-xml-19980210`.

[CCD+98] S. Ceri, S. Comai, E. Damiani, P. Fraternalli, S. Paraboshi, and L. Tanca. XML-GL: a Graphical Language for Querying and Restructuring XML Documents, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xml-gl.html`.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML, 1998. Available at `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`.

[GW98] R. Goldman and J. Widom. Interactive query and search in semistructured databases. In *International Workshop on the Web and Databases*, Valencia, Spain, 1998.

[IKK98] H. Ishikawa, K. Kubota, and Y. Kanemasa. XQL: A Query Language for XML Data, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/flab.txt`.

[LPVV99] B. Ludäscher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View Definition and DTD inference for XML. In *Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1999.

[PV99] Y. Papakonstantinou and P. Velikhov. Enchancing Semistructured Data Mediators with Document Type Definitions. In *Proceedings of the 15th International Conference on Data Engineering*. IEEE Computer Society Press, 1999.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL), 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.