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 a 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 DTDs 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. Automatic generation of query results is discussed in Section 6. In Section 7, we review related work and 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

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

to d, and the root element of d is the outermost element of x. Given a DTD, without a root, which specifies elements $e_1, \ldots, e_n$, we can create a DTD with the root element "root" by adding <! ELEMENT root $(e_1|e_2|\ldots|e_n)*$ >. Thus, we can assume, without loss of generality, that all DTDs have a root element. Below we give an example of a DTD with a root. This DTD is used in the sequel.

```
<!ELEMENT root        (course+)>
<!ELEMENT course      (name, teacher, time+)>
<!ELEMENT teacher     (name)>
<!ELEMENT time       (day, start_hour, finish_hour)>
<!ELEMENT day         (#PCDATA)>
<!ELEMENT start_hour  (#PCDATA)>
<!ELEMENT finish_hour (#PCDATA)>
<!ELEMENT name        (#PCDATA)>
```

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

Definition 2.1 (Catalog) *A* catalog *is a triple,* $< d, id, X >$*, where* d *is a DTD,* id *is a unique identifier 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 (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 to discover its structure.

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 that $\mathcal{L}$ 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 tree, i.e., every node in* N *is reachable from* r*;*

[1]In general, a document may be a graph, and not a tree, as a result of ID and IDREF attributes. However, we view documents as trees since we can express the graph structure in a query using several forms and internode constraints.

- $l : N \rightarrow \mathcal{L}$ *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*is a* terminal mapping*, i.e.,* α *maps terminal nodes to atomic values 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 she 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 earlier. The query computes the courses not taught by Dr. Jekyll.

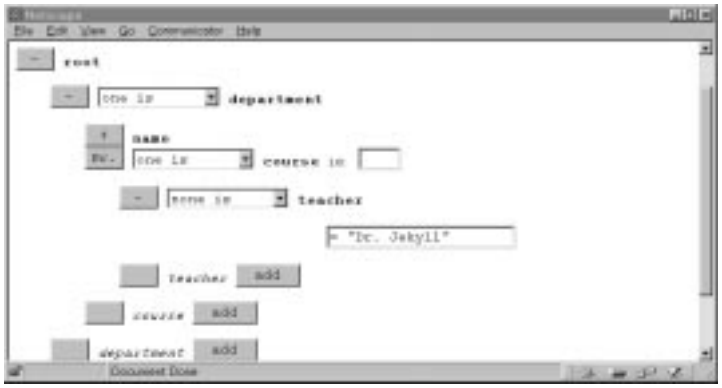

Figure 1: EquiX Query

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 as 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*, to their names. The result document is automatically created by the query evaluation process.

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.

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. The semantics of aggregation, e.g., the grouping elements, is discussed in Section 5. EquiX ignores data that violate the type constraint imposed by the aggregation function.

The aggregation function *max* (*min*) enables the user to find the maximal (minimal) value of an element e. However, it does not allow the user to easily find the values of another element e' for which e gets its maximal (minimal) value. For example, given a database containing student names and grades, the maximal grade can be found using the *max* function. However, it is difficult to

find the name of the student that received the maximal grade. When querying in SQL, in order to retrieve this type of information, it is necessary to use nested queries. This is not intuitive and is difficult for a naive user. Therefore, we have included in EquiX two *aggregation constraints*, denoted max^c and min^c. Typing an aggregation constraint in a field, constrains the field to its maximal or minimal value. Thus the effect of a nested query is acheived easily.

3.3 Constraining Non-Terminal Elements

Non-terminal 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 to define the semantics of queries. We define a query tree below.

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

- *cond is a mapping of terminal nodes to conditions;*
- *quant maps each edge to a quantifier in* $\{\exists, \forall, \neg\exists, \neg\forall\}$;
- *agg is a partial mapping of nodes to aggregation functions in* {count, sum, avg, max, min} *or to aggregation constraints in* $\{min^c, max^c\}$;
- $P \subseteq N$ *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) implies 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. Note that for terminal nodes that are not constrained in the form, *cond* is defined as **true**. Quantification and aggregation of elements are represented by the mappings *quant* and *agg*, respectively. Each edge in the query tree is mapped by *quant* to one of the quantifiers.

In the general case, a query may be created from several forms, thus producing several query trees. We reduce the problem of querying using several query trees to querying using a single tree. Suppose that the user creates a query from k query

forms, deriving trees $Q_1, \ldots, Q_k$. Suppose that the root of Q_i is r_i. We create a new query tree Q whose root is defined as <!ELEMENT r $(r_1, \ldots, r_k)$ >. This root is a *virtual root* since it does not actually appear in any DTD. We create a set of XML documents to be queried in the following way. For each set of documents $D_1, \ldots, D_k$, such that D_i is a document in the catalog of the DTD of Q_i, we create a document D that has r as a root and r_i as children of r. Thus, instead of evaluating $Q_1, \ldots, Q_k$ on $D_1, \ldots, D_k$, we can evaluate Q on D. Therefore, we assume in the sequel that a query always yields a single query tree.

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 in EquiX than in SQL. In SQL, projected attributes and aggregation functions are expressed in the SELECT clause. In our formalism, aggregation is represented by the mapping *agg*, and the nodes that are included in the result are represented by P.

In this section we present the semantics of an EquiX query. To simplify we assume that there are no aggregation functions, aggregation constraints or variables in the query. We discuss an appropriate extension for aggregation functions and constraints in Section 5. Variables are necessary for internode constaints. Thus, variables are required in order to constrain two nodes to have the same value. However, when comparing a single node to a constant, constants are not necessary. Extending the semantics to deal with variables involves some subtle problems. First, when using variables in a query, where do they get their bindings? Second, how do the quantifications effect the meaning of a query with variables? To solve the first problem, EquiX requires the user to specify one node in the query where a given variable gets its bindings. In all other nodes, the variable may be used as part of a constraint. We have also solved the second problem by appropriately extending the semantics presented in this section. However, we will not present this extension here for lack of space.

When describing the semantics of a query in a relational database language, such as SQL or Datalog, the term *matching* can be used. The result of evaluating a query are all the tuples that match the schemas mentioned in the query and satisfy the constraints. We describe the semantics of an EquiX query in a similar fashion. We first present an auxillary definition.

We can divide the nodes in a query tree into two different classes. A node n is a *negative node* if the path from the root to n has an odd number of negated quantifiers. (i.e., $\neg\exists$, or $\neg\forall$). Otherwise, n is a *positive node*. Note that negation in a query adds implicit "or"s to the query. If we were to push the negations

in the quantifiers down the tree, we would derive an *and-or tree*. Intuitively, negative nodes correspond to "or" nodes and positive nodes correspond to "and" nodes. Thus, beneath a negative node we must satisfy at least one child, while beneath a positive node we must satisfy all children. We define a *set of support* for a query Q to be a set of nodes that must be satisfied in order to satisfy Q. Observe that a given query may have many different sets of support.

Definition 4.2 (Set of Support) *Consider the query tree Q with nodes N_Q and root r_Q. We say that $\mathcal{S} \subseteq N_Q$ is a* set of support *in Q if the following holds:*

1. *$r_Q \in \mathcal{S}$;*
2. *if $n \in \mathcal{S}$, then $p(n) \in \mathcal{S}$, where $p(n)$ denotes the parent of n;*
3. *if $n \in \mathcal{S}$ is a positive node, then all the children of n are in S;*
4. *if $n \in \mathcal{S}$ is a negative node, then at least one child of n is in S.*

We say that n is a node of support *if n is in $\mathcal{S}$.*

Note that Condition 1 insures that the root is a node of support, i.e. it must be satisfied. Condition 2 insures that the set of nodes has a tree-like structure. Condition 3 insures that beneath positive nodes ("and" nodes) all children must be satisfied, and Condition 4 insures that beneath negative nodes ("or" nodes) at least one child must be satisfied.

Consider an edge (n, m) with quantifier $quant(n, m)$ in a query. If m is a positive node, then in order to satisfy m, we must satisfy $quant(n, m)$. However, observe that if m is a negative node, then satisfying m actually entails satisfying the quantification $\neg quant(n, m)$. Thus, we say that the *virtual quantifier* of an edge (n, m), denoted $vquant(n, m)$, is $quant(n, m)$ if m is a positive node and $\neg quant(n, m)$ if m is a negative node.

Definition 4.3 (Matching) *Let $D = (G_D, \alpha)$ be a document tree, with $G_D = (N_D, E_D, r_D, l_D)$, and let $Q = (G_Q, cond, quant, agg, P)$ be a query tree, with $G_Q = (N_Q, E_Q, r_Q, l_Q)$. Suppose that $\mathcal{S}$ is a set of support of Q. Suppose that μ is a function $\mu : N_Q \to 2^{N_D}$, such that the following hold*

1. *$\mu(r_Q) = \{r_D\}$;*
2. *if $n' \in \mu(n)$, then $l_D(n') = l_Q(n)$;*
3. *if $n' \in \mu(n)$, then $p(n') \in \mu(p(n))$;*
4. *if $(n, m) \in E_Q$, $n' \in \mu(n)$, and $m \in \mathcal{S}$, then the following holds:*
   - *if $vquant(n, m) \in \{\exists, \neg\forall\}$, then $\mu(m)$ contains exactly one child of n'.*
   - *if $vquant(n, m) \in \{\forall, \neg\exists\}$ then $\mu(m)$ contains all the children of n'.*

Then μ is a matching *of D to Q w.r.t. (with respect to) $\mathcal{S}$.*

Note that Condition 1 requires that the root of the document is matched to the root of the query, Condition 2 insures that matching nodes have the same label, and Condition 3 requires matchings to have a tree-like structure. Condition 4 compels μ to satisfy the quantification constraints of the query for all the nodes in the set of support $\mathcal{S}$.

We now define when a matching satisfies a document. As in the relational case, this will enable us to describe the result of evaluating EquiX queries. We first present an auxillary definition of the satisfaction of a node of a document tree with respect to a matching and a query. This definition is recursive on the structure of the query.

Definition 4.4 (Satisfaction of a Node) *Let D be a document tree with nodes N_D and terminal mapping α. Let Q be a query tree with nodes N_Q, quantification quant, and conditions cond. Let $\mathcal{S}$ be a set of support and let μ be a matching of D to Q w.r.t. $\mathcal{S}$. Suppose that $n' \in \mu(n)$ for some $n' \in N_D$ and $n \in \mathcal{S}$. We say that n'* satisfies *n* w.r.t. *μ, denoted $n' \models_{\mu,\mathcal{S}} n$, if the following conditions hold:*

- *if n is a terminal node, then $cond(n)$ is true with respect to $\alpha(n')$;*
- *if n is an internal node, then for each child m of n, such that $m \in \mathcal{S}$, the following holds:*
  1. *if $quant(n,m) \in \{\exists, \forall\}$, then for all $m' \in \mu(m)$ it holds that $m' \models_{\mu,\mathcal{S}} m$;*
  2. *if $quant(n,m) \in \{\neg\exists, \neg\forall\}$, then for all $m' \in \mu(m)$ it holds that $m' \not\models_{\mu,\mathcal{S}} m$;*

Definition 4.5 (Satisfying Matching) *Consider a document tree D with root r_D, and query tree Q with root r_Q. Let μ be a matching of D to Q w.r.t. $\mathcal{S}$. Then μ is a* satisfying matching, *denoted $D \models_{\mu,\mathcal{S}} Q$ if μ is a matching, such that $r_D \models_{\mu,\mathcal{S}} r_Q$.*

Recall that a query is created by exploring a DTD that relates to a catalog. This catalog has a list of XML documents associated with it in the database. Having now defined when a matching satsfies a query, we specify the output of evaluating a query on an XML document. The result of evaluating a query will be the ouptut of evaluating the query on each document in the queried catalog.

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, intuitively, the result of evaluating a query on a document is a subtree of the document. Given a query and document tree, it is only necessary to specify which nodes of the document tree are *marked*, i.e., appear in the output.

Consider a document D with nodes N_D and a query Q with projected nodes P_Q. Let $\mathcal{M}$ be the set of all pairs of a satisfying mapping μ and a set of support $\mathcal{S}$ of Q, such that $D \models_{\mu,\mathcal{S}} Q$. Formally, $\mathcal{M} := \{(\mu, \mathcal{S}) \mid D \models_{\mu,\mathcal{S}} Q\}$. We consider two options for marking nodes of D.

Association Marking Intuitively, we mark all the nodes in the document that are associated with a marked node in the query (i.e., a node in P) by a satisfying matching. This is similar to finding the output of an SQL query, since all elements that match the query and are included in the projection appear in the result. Formally, we define the set of nodes marked in the following way:

$$AM(Q, D) = \{n' \in N_D \mid (\exists \mu)(\exists \mathcal{S})(\exists n)((\mu, \mathcal{S}) \in \mathcal{M} \ \wedge \ n \in P \ \wedge \ n' \in \mu(n))\}$$

Witness Marking Once again we mark nodes in the document that are associated with a marked node in the query by a satisfying matching. However, we restrict ourselves to marking only such nodes that "bear witness" to the satisfaction of the query. Recall that negation in a query can create an implicit "or." Thus, some nodes bear witness to the satisfaction of the "or" (i.e., those in the set of support) while others do not. We only mark the nodes of the first type. This option is more intuitive for a naive user, since only values that satisfy the query will appear in the result. Thus, the set of nodes marked is defined as:

$$WM(Q, D) = \{n' \in N_D \mid (\exists \mu)(\exists \mathcal{S})(\exists n)((\mu, \mathcal{S}) \in \mathcal{M} \ \wedge \ n \in P \ \wedge \ n' \in \mu(n) \ \wedge \ n \in \mathcal{S})\}$$

We now define the *association-marking result tree* $R_{AM}(Q, D)$ of evaluating a query Q on an XML document D. A *witness-marking result tree* $R_{WM}(Q, D)$ can be defined similarly.

Definition 4.6 (Association-Marking Result Tree) *Let $D = (G_D, \alpha_D)$ be a document tree, where $G_D = (N_D, E_D, r_D, l_D)$, and let Q be a query. The* association-marking result tree *of evaluating Q on D is the document tree $R_{AM}(Q, D) = (G_R, \alpha_R)$, where $G_R = (N_R, E_R, r_R, l_R)$ and the following hold:*

- $N_R := AM(Q, D) \ \cup \ descendants(AM(Q, D)) \ \cup \ ancestors(AM(Q, D))$;
- $r_R = r_D$;
- E_R, l_R, *and* α_R *are* E_D, l_D, *and* α_D, *respectively, restricted to* N_R.

Note that $descendants(N)$ is the set of descendants of N and $ancestors(N)$ is the set of ancestors of N. We add the set of descendents to insure that there is actual information (i.e., PCDATA) in the result. Adding the set of ancestors insures that the result is a rooted tree, and thus, is in the same data model as the input. We say that R is a *result tree* if R is an association-marking result tree or a witness-marking result tree. For queries without negation, association-marking result trees and witness-marking result trees are always identical, as the proposition below claims.

Proposition 4.7 *Let Q be a query tree and D be a document tree. Suppose that Q does not contain any edges quantified by a negated quantifier. Then $R_{AM}(Q, D) = R_{WM}(Q, D)$.*

5 Query Evaluation

We present an algorithm for evaluating a query on a document. To simplify, we assume that the query contains neither variables nor aggregation. It is straightforward to extend our algorithm to deal with variables and aggregation. The algorithm presented computes the witness marking of a given document with respect to a given query. The association marking of a document tree can be computed similarly by making appropriate modifications to the algorithm.

It would seem that computing a witness marking should be computationally expensive. There is an exponential number of sets of support and an exponential number of matchings. Thus, it would seem difficult to find a polynomial algorithm for this problem (where both the query and the database are considered as the input). Quantifiers in the query add an additional source of complexity to the query processing. Roughly speaking, however, query evaluation in this case is analogous to evaluating a first-order query that can be written using only two variables. Therefore, using dynamic programming we can in fact derive an algorithm that runs in polynomial time, even when the query is considered part of the input (i.e., combined complexity).

In Figure 2 we present a polynomial algorithm for computing the witness marking of a document, given a query. This algorithm uses the procedure shown in Figure 3 in order to check if a document node satisfies a query node. Note that $path(n)$ is the sequence of element names on the path from the root of the query to n. The predicate $pos(n)$ is true if n is a positive node and is false otherwise. Note also that *sat_array* is an array of size $N_Q \times N_D$ of boolean values, where N_Q are the nodes in the query and N_D are the nodes in the document. The value in *sat_array*[n,n'] will be true if n' satisfies n for some pair of a matching and a set of support. Observe that in Figure 2 we order the nodes by descending depth. This insures that when Satisfies(n',n,*sat_array*) is called, the array *sat_array* is already updated for all the children of n' and n. The algorithm Witness_Marking does not explicitly create any matchings or sets of support. However, the following the following lemma holds.

Lemma 5.1 *Let Q be a query tree with projected nodes P and let D be a document tree. A node n' in the document tree is returned by* Witness_Marking *if and only if there is a set of support $\mathcal{S}$, a matching μ, and a query node n, such that the following hold:*

- *$n \in P$, i.e., n is a projected node in the query;*
- *$n' \in \mu(n)$, i.e., n' is associatd with n in the matching;*
- *$n \in \mathcal{S}$, i.e., n is a node of support;*
- *$D \models_{\mu,\mathcal{S}} Q$, i.e., D satisfies Q w.r.t. μ and $\mathcal{S}$.*

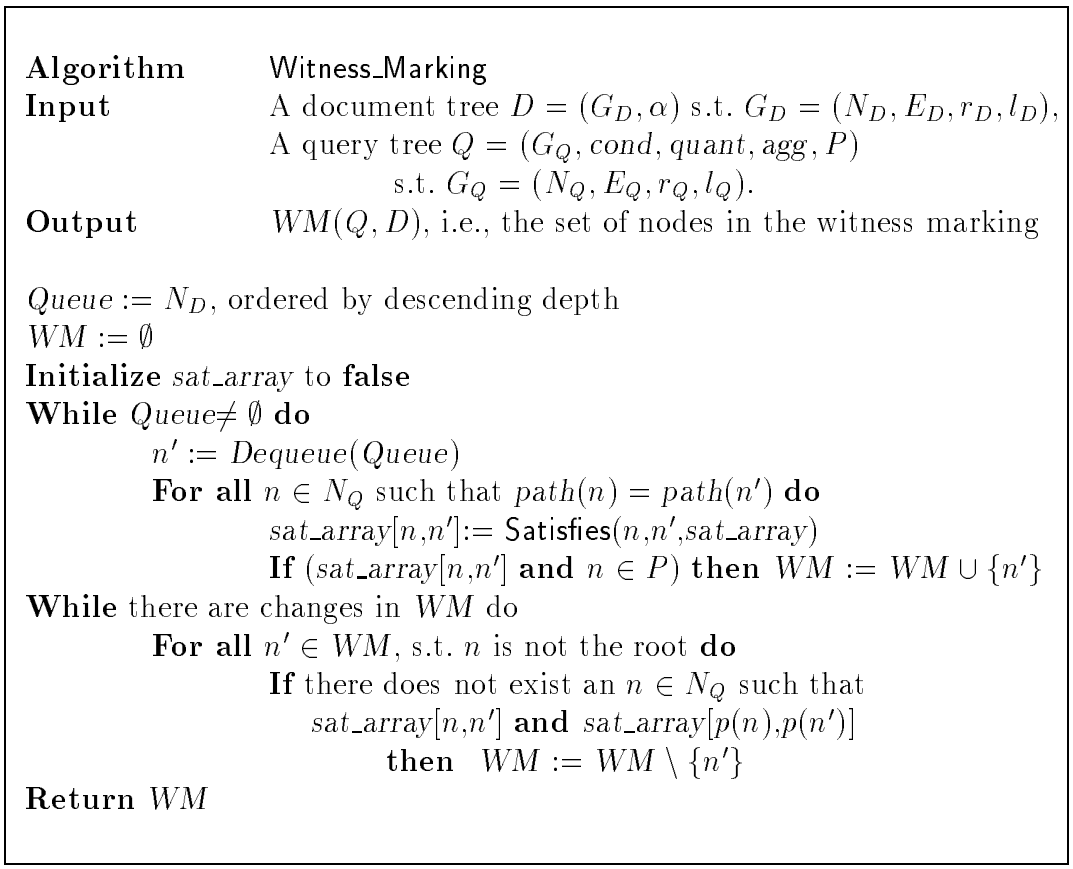

Figure 2: Witness Marking Evaluation Algorithm

Theorem 5.2 *Given a query tree Q with nodes N_Q and a document tree D with nodes N_D, the result of* Witness_Marking*(Q,D) is equal to $WM(Q, D)$. In addition,* Witness_Marking *runs in time $O(|N_Q| \cdot |N_D|^2)$.*

As stated above, Witness_Marking 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.

Procedure Satisfies(n,n',sat_array)
Input A query node n
A document node n'
A partially filled array sat_array such that sat_array[m,m'] is true if m' satisfies m
Output **true** if n' satisfies n and **false** otherwise

If n is a terminal node **then**
    $status$:= the value of $cond(n)$ with respect to $\alpha(n')$
Else
    Let M be the set of children of n in Q;
    For each $m \in M$ do:
        Let M' be the set of children, m', of n' in D such that $l_D(m) = l_Q(m')$
        Switch $quant((n,m))$:
            $\exists$: $status_m := \bigvee_{m' \in M'} sat\_array[m,m']$
            $\forall$: $status_m := \bigwedge_{m' \in M'} sat\_array[m,m']$
            $\neg\exists$: $status_m := \neg(\bigvee_{m' \in M'} sat\_array[m,m'])$
            $\neg\forall$: $status_m := \neg(\bigwedge_{m' \in M'} sat\_array[m,m'])$
    $status := \bigwedge_{m \in M} status_m$
If ($pos(n)$ **and** $status$) **Return true**
Else If (**not** $pos(n)$ **and not** $status$) **Return true**
Else Return false

Figure 3: Satisfaction of a Node Procedure

6 Creating Query Results

The result of an EquiX query is a catalog. In this section we define the XML documents and the DTDs that are the result of query processing. We do not require the user define explicity the structure of the result. Thus, querying is made easier.

When applying a query Q to an XML document D, a result tree R is created. The result tree is actually a document tree created by projecting out some of the nodes from the document. Thus, the resulting document is an XML document whose parse tree is R. Note that we create the resulting document while retaining the structure of the original document with respect to the elements projected onto. Queries are applied to several documents, and thus, a set of documents is created.

We now consider the problem of creating a DTD for the resulting catalog, called a *result DTD*. A result DTD is a DTD to which all resulting documents conform. This DTD is based on the definitions in the *original DTD*, i.e., the DTD from which the query was created. The result DTD must contain a definition for all elements that might appear in the result. Consider a document D and query Q with the labeling function l_Q and the set of projected nodes P. The elements that may appear in the result are either labels of each node n' in D that satisfies a node n in P or labels of ancestors or descendants of node n'. We can find these elements in two steps.

We *first* extend P to include all the ancestors and descendants of P in Q. We call this extension P^+. Clearly all the elements that are labels of nodes in P^+ must be defined in the result DTD.

Consider a node n' that corresponds to a node n in the query. Note that while the ancestors n' correspond to nodes in the query, some of the descendants may not. This follows since the result tree contains all the descendents of marked nodes in the document tree, even if Q is not be explored to the full depth of the document. Thus, our *second* step is to find the labels (i.e., element names) of these nodes. This can be done by recursively examining the original DTD in the following fashion. We say that an element e' is a descendant of an element e in a given DTD if e' appears in the content definition of e, or if e' appears in the content definition of an element e'' that is a descendent of e. (The *content definition* of an element e is φ if e is defined as <!ELEMENT e φ >.) Thus, the element names of the descendents of n' are the descendents of $l_Q(n)$ in the original DTD.

We can now formally describe the set of elements $\mathcal{E}$ that must be defined in the result DTD. This set is actually the union of two sets that we define as:

- $\mathcal{E}_1 := \{e|(\exists n)(l_Q(n) = e \wedge\ n \in P^+)\}$, i.e., the labels of nodes projected, or their ancestors and descendants in Q;
- $\mathcal{E}_2 := \{e|(\exists n)(n \in P\ \wedge\ e \in \mathit{descendants}(l_Q(n)))\}$, i.e., the labels of descendents according to the original DTD.

Then the set of elements that must be defined in the result DTD is $\mathcal{E} := \mathcal{E}_1 \cup \mathcal{E}_2$.

We must also define the attributes appearing in the result DTD. These attributes are exactly 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.

We present an algorithm for creating the content definition of an element $e \in \mathcal{E}$. Intuitively, any elements that will not appear in the result tree of a query must be removed from the original DTD in order to form the result DTD. Quantification in a query can also effect the result DTD. Recall that beneath

negative nodes it is sufficient to satisfy only one child. However, we have no a priori knowledge which child will be satisfied, and thus, marked. Therefore, we must relax the original DTD to fit the result.

The algorithm for creating result content definitions is presented in Figure 4. This algorithm uses the apreocedure presented in Figure 5 in order to simplify the content definition it creates. The actual result DTD is created by gathering the content definitions for all $e \in \mathcal{E}$ and adding the attribute definitions. It is easy to see that the algorithm in fact produces a DTD appropriate for the result and that it runs in polynomial time.

Theorem 6.1 *Consider a query Q formed from a DTD $\mathcal{T}$ and a document D. Let R be the result tree created by evaluating Q on D. Let $\mathcal{T}'$ be the result DTD formed by the process described above. Then R conforms to $\mathcal{T}'$ and the process of formulating $\mathcal{T}'$ is polynomial in the size of Q and $\mathcal{T}$.*

Algorithm Create_DTD_Definition
Input An element $e \in \mathcal{E}$
The extended set P^+ of projected nodes
The original DTD $\mathcal{T}$
Output The definition of e in the result DTD

/* Let φ denote the content definition of e in $\mathcal{T}$ */
If $e \in \mathcal{E}_2$ **then** *Content* := φ
Else *Content* := ϵ
For all nodes $n \in P^+$ with label e, (i.e., with $l(n) = e$) **do**
    $\varphi_n := \varphi$
    If n is not a leaf **then**
        For all elements e_i in φ **do**
            If there is a child of n in P^+ with the label e_i **then**
                Replace all occurrences of e_i in φ_n with e_i?
            Else
                Replace all occurrences of e_i in φ_n with ϵ
    Content := *Content* | φ_n
If Simplify(*Content*) = ϵ **then**
    Return <!ELEMENT e EMPTY >
Else Return <!ELEMENT e Simplify(*Content*) >.

Figure 4: DTD Definition Generation Algorithm

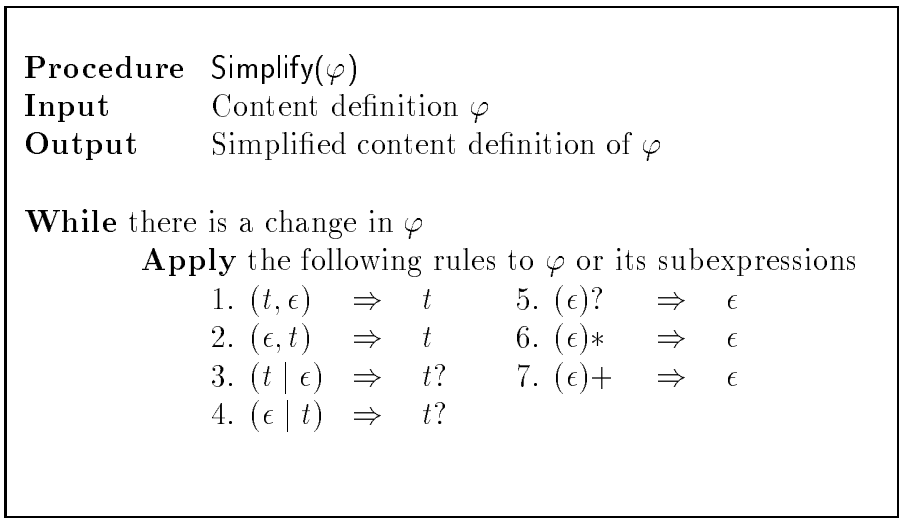

Figure 5: Definition Simplifying Algorithm

We say that a result DTD $\mathcal{T}$ is *tight* ([PV99]) if there does not exists a result DTD $\mathcal{T}'$, such that the set of documents conforming to $\mathcal{T}'$ is properly contained in the set of documents conforming to $\mathcal{T}$. Clearly, tightness is a desirable property, since a tighter result DTD describes more precisely the resulting documents. Our algorithm does not necessarily create a tight result DTD. However, we show that there are queries and DTDs for which a tight result DTD must be exponential in the size of the original DTD.

Theorem 6.2 *There is a query Q derived from a DTD $\mathcal{T}$, such that if $\mathcal{T}'$ is a tight result DTD of Q, then $\mathcal{T}'$ is of size $O(|\mathcal{T}|!)$.*

Proof. Consider a DTD containing the following definition:

$$<!\text{ELEMENT } r\ (a_1, \ldots, a_k)* >$$

We create a query tree with root r and children $a_1, \ldots, a_k$. Suppose that all the nodes a_i are projected onto and all the edges in the tree (i.e., those from r to a_i) are existentially quantified. In all result documents the elements a_i must appear. However, they can appear in any order. Thus, a tight result DTD must consider $k!$ different orderings of the elements, proving the claim. □

Note that our algorithm will produce a compact result DTD for the example in the proof. The content definition of r will be $<!\text{ELEMENT } r\ (a_1?, \ldots, a_k?)* >$. Observe that an exponential blowup of the original DTD is undesirable for two reasons. First, creating such a DTD is intractable. Second, if the result DTD is of exponential size, then it is difficult for a user to re-query previous results. Thus, our algorithm for creating result DTDs actually returns a convenient DTD, although it is not always tight.

7 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. 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, that allows aggregation, quantification and negation. The results created fit the data model. Thus, iterative querying is possible ([GW98]). A large class of EquiX queries can be computed in polynomial time, even when the query is considered part of the input. Thus, querying with EquiX is an efficient process.

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] is 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. We are also planning to incorporate ordering constraints into EquiX and investigate how this affects the complexity of querying. Identifying classes of DTDs and queries for which tight result DTDs of polynomial size exist is another area for future research.

We are currently in the process of implementing EquiX. The implementation naturally gives rise to optimization problems, such as indexing and view usability. We are planning on investigating these problems.

Acknowledgments

This research was supported in part by the Esprit Long Term Research Project 22469 "Foundations of Data Warehouse Quality" (DWQ), and by Grants 8528-95-1 and 9481-1-98 of the Israeli Ministry of Science.

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`.