EquiX—A Search and Query Language for XML

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

Abstract

EquiX is a search language for XML that combines the power of querying with the simplicity of searching. Requirements for such languages are discussed and it is shown that EquiX meets the necessary criteria. Both a graph-based abstract syntax and a formal concrete syntax are presented for EquiX queries. In addition, the semantics is defined and an evaluation algorithm is presented. The evaluation algorithm is polynomial under combined complexity.

EquiX combines pattern matching, quantification and logical expressions to query both the data and meta-data of XML documents. The result of a query in EquiX is a set of XML documents. A DTD describing the result documents is derived automatically from the query.

1 Introduction

The widespread use of the World-Wide Web has given rise to a plethora of simple query processors, commonly called search engines. Search engines query a database of semi-structured data, namely HTML pages. Currently, search engines cannot be used to query the meta-data content in such pages. Only the data can be queried. For example, one can use a search engine to find pages containing the word "villain". However, it is difficult to obtain only pages in which villain appears in the context of a character in a Wild West movie. More and more XML pages are finding their way onto the Web. Thus, it is becoming increasingly important to be able to query both the data and the meta-data content of the pages on the Web. We propose a language for querying (or searching) the Web that fills this void.

Search engines can be viewed as simple query processors. The query language of most search engines is rather restricted. Both traditional database query languages, such as SQL, and newly proposed languages, such as XQL [RLS98], XML-QL [DFF+98] and Xmas [BLP+98, LPVV99], are much richer than the query language of most search engines. However, the limited expressiveness of search engines appears to be an advantage in the context of the Web. Many Internet users are not familiar with database concepts and find it hard to formulate SQL queries. In comparison, when it comes to using search engines, experience has proven that even novice Internet users can easily ask queries using a search engine. It is likely that this is true because of the inherent simplicity of the search-engine query languages.

Consequently, an apparent disadvantage of search-engine languages is really an advantage when it comes to querying the Web. Thus, it is imperative to first understand the requirements of a query language for the Web, before attempting to design such a language. We believe that the Web gives rise to a new concept in query languages, namely *search languages*. We will present design criteria for search languages.

*Institute for Computer Science, The Hebrew University, Jerusalem 91904, Israel.
†Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, EH14 4AS.
‡Department of Computer Science, K. U. Leuven Celestijnenlaan 200A, B-3001, Heverlee, Belgium.

As its name implies, a search language is a language that can be used to search for data. We differentiate between the terms *search* and *query*. Roughly speaking, a search is an imprecise process in which the user guesses the content of the document that she requires. Querying is a precise process in which the user specifies exactly the information she is seeking. In this paper we define a language that has both searching and querying capabilities. We call a language that allows both searching and querying a search language.

We call a query written in a search language a *search query* and the query result a *search result*. Similarly, we call a query processor for a search language a *search processor*. From analyzing popular search engines, one can define a set of criteria that should guide the design of a search language and processor. We present such criteria below.

1. **Format of Results:** A search result of a search query should be either a set of documents (pages) or sections of documents that satisfy the query. In general, when searching, the user is simply interested in *finding* information. Thus, a search query need not perform restructuring of documents to compute results. This simplifies the formulation of a search query since the format of the result need not be specified.

2. **Pattern Matching:** A search language should allow some level of pattern matching both on the data and meta-data. Clearly, pattern matching on the data is a convenient way of specifying search requirements. Pattern matching on the meta-data allows a user to formulate a search query without knowing the exact structure of the document. In the context of searching, it is unlikely that the user will be aware of the exact structure of the document that she is seeking.

3. **Quantification:** Many search languages currently implemented on the Web allow the user to specify quantifications in search queries. For example, the search query "`+Wild -West`", according to the semantics of many of the search engines found on the Web, requests documents in which the word "`Wild`" appears (i.e., exists) and the word "`West`" does not appear (i.e., not exists). The ability to specify quantifications should be extended to allow quantifications in querying the meta-data.

4. **Logical Expressions:** Many search engines allow the user to specify logical expressions in their search languages, such as conjunctions and disjunctions of conditions. This should be extended to enable the user to use logical expressions in querying the meta-data.

5. **Iterative Searching Ability:** The result of a search query is generally very large. Many times a result may contain hundreds, if not thousands, of documents. Users generally do not wish to sift through many documents in order to find the information that they require. Thus, it is a useful feature for a search processor to allow requerying of previous results. This enables users to search for the desired information iteratively, until such information is found.

6. **Polynomial Time:** The database over which search queries are computed is large and is constantly growing. Hence, it is desirable for a search query to be computable in polynomial time under combined complexity (i.e., when both the query and the database are part of the input).

When designing a search language, there is an additional requirement that is more difficult to define scientifically. A search language should be *easy to use*. We present our final criterion.

7. **Simplicity:** A search language should be simple to use. One should be able to formulate queries easily and the queries, once formulated, should be intuitively understandable.

The definition of requirements for a search language is interesting in itself. In this paper we present a specific language, namely EquiX, that fulfills the requirements 1 through 6. From our experience, we have found EquiX search queries to be intuitively understandable. Thus, we believe that EquiX satisfies the additional language requirement of simplicity. EquiX is rather unique in that it combines both polynomial query evaluation (under combined complexity) with several powerful querying abilities. In EquiX, both quantification and negation can be used. Regular expressions can also be used on the data of an XML document. In an extension toEquiX we allow aggregation on the data and a limited class of regular expressions on the metadata. Both searching and querying can be performed using the EquiX language. EquiX also simplifies the querying process by automatically generating both the format of the result and a corresponding DTD.

This paper extends previous work [CKK+99, CKK+00]. In Section 2 we present a data model for XML documents. Both the concrete and abstract syntax for EquiX queries are described in Section 3. In Section 4 we define the semantics of EquiX, and in Section 5 a polynomial algorithm for evaluating EquiX queries is presented. A procedure for computing a result DTD is presented in Section 6. In Section 7 we present some extensions to our language and in Section 8 we conclude. We present proofs of theorems in Appendix A.

2 Data Model

We define a data model for querying XML documents [BPSM98]. At first, we assume that each XML document has a given DTD. In Section 7 we will relax this assumption. The term *element* will be used to refer to a particular occurrence of an element in a document. The term *element name* will refer to the name of an element and thus, may appear many times in a document. Similarly we use *attribute* to refer to a particular occurrence of an attribute and *attribute name* to refer to its name. At times, we will blur the distinction between these terms when the meaning is clear from the context.

We introduce some necessary notation. A *directed tree* over a set of nodes N is a pair $T = (N, E)$ where $E \subseteq N \times N$ and E defines a tree-structure. We say that the edge (n, n') is *incident from* n and *incident to* n'. Note that in a tree, there is at most one edge incident to any given node. We assume throughout this paper that all trees are finite. The *root* of a directed tree is the node $r \in N$, such that every node in N is reachable from r in T. We denote a rooted directed tree as a triple $T = (N, E, r)$.

An XML document contains both data (i.e., atomic values) and meta-data (i.e., elements and attributes). The relationships between data and meta-data, (and between meta-data and meta-data) are reflected in a document by use of nesting.

We will represent an XML document by a directed tree with a labeling function. The data and meta-data in a document correspond to nodes in the tree with appropriate labels. Nodes corresponding to meta-data are *complex nodes* while nodes corresponding to data are *atomic nodes*. The relationships in a document are represented by edges in the tree. In this fashion, an XML document is represented by its parse tree.

Note that using `ID` and `IDREF` attributes one can represent additional relationships between values. When considering these relationships, a document may no longer be represented by a tree. In the sequel we will utilize `ID` and `IDREF` attributes to answer search queries.

In general, a parsed XML document need not be a rooted tree. An XML document that gives rise to a rooted tree is said to be *rooted*. The element that corresponds to the root of the tree is called the *root element*. Given an XML document that is not rooted, one can create a rooted document by adding a new element to the document and placing its opening tag at the beginning of the document, and its closing tag at the end of the document. This new element

```
<!ELEMENT movieInfo   (movie+,actor+)>
<!ELEMENT movie       (descr,title,character+)>
<!ELEMENT actor       (name)>
<!ATTLIST actor
          id     ID      #REQUIRED>
<!ELEMENT descr       (#PCDATA)>
<!ELEMENT title       (#PCDATA)>
<!ELEMENT name        (#PCDATA)>
<!ELEMENT character   EMPTY>
<!ATTLIST character
          role   CDATA   #REQUIRED
          star   IDREF   #REQUIRED>
```

Figure 1: DTD describing movie information

will be the root element of the new document. With little effort we can adjust the DTD of the original document to create a new DTD that the new document will conform to. Thus, we assume without loss of generality that all XML documents in a database are rooted.

We now give a formal definition of an XML document. We assume that there is an infinite set $\mathcal{A}$ of atoms and infinite set $\mathcal{L}$ of labels.

Definition 2.1 (XML Document) *An* XML document *is a pair* $X = (T, l)$ *such that*

- $T = (N, E, r)$ *is a rooted directed tree;*[1]
- $l: N \to \mathcal{L} \cup \mathcal{A}$ *is a labeling function that associates each complex node with a value in* $\mathcal{L}$ *and each atomic node with a value in* $\mathcal{A}$.

We assume that each DTD has a designated element name, called the *root element name* of the DTD. Consider a DTD d with a root element name e. We say that a document $X = (T, l)$ with root r *strictly conforms to* d if

1. the document X conforms to d (in the usual way [BPSM98]) and
2. the function l assigns the label e to the root r (i.e., $l(r) = e$).

The DTD in Figure 1 with root element name `movieInfo` describes information about movies. In Figure 2 an XML document containing movie information is depicted. This document strictly conforms to the DTD resented above. Note that the nodes in Figure 2 are numbered. The numbering is for convenient reference and is not part of the data model.

A *catalog* is a pair $C = (d, S)$ where d is a DTD and S is a set of XML documents, each of which strictly conforms to d. A *database* is a set of catalogs. Note the similarity of this definition to the relational model where a database is a set of tuples conforming to given relation schemes.

This data model is natural and has useful characteristics. Our assumption that each XML document conforms to a given DTD implies that the documents are of a partially known structure. We can display this knowledge for the benefit of the user. Thus, the task of finding information in a database does not require a preliminary step of querying the database to discover its structure.

[1] Note that an XML document is a sequence of characters. Thus, to properly model the ordering of elements in a document, an ordering function on the children of a node should be introduced. For simplicity of exposition we chose to omit this in the paper.

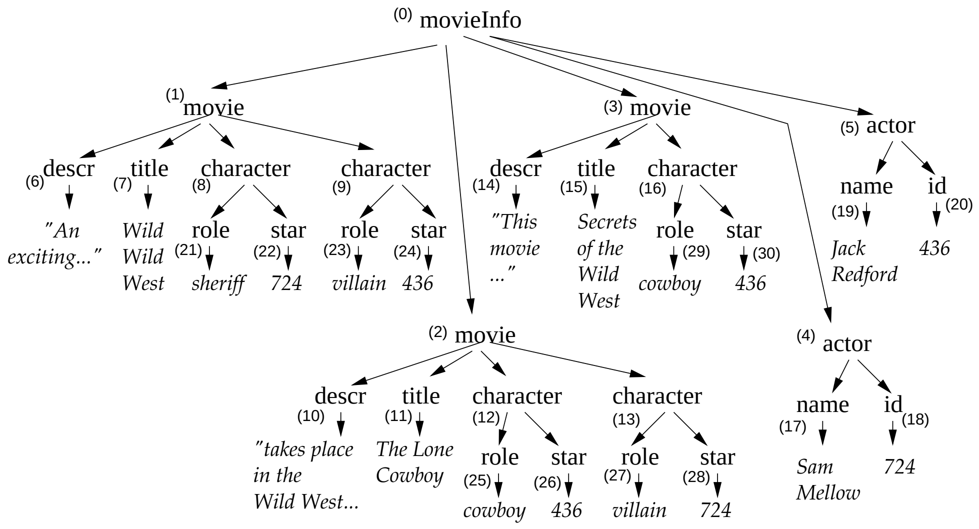

Figure 2: An XML Document

3 Search Query Syntax

In this section we present both a concrete and an abstract syntax for EquiX search queries. A search query written in the concrete syntax is a *concrete query* and a search query written in the abstract syntax is an *abstract query*.

3.1 Concrete Query Syntax

The concrete syntax is described informally as part of the graphical user interface currently implemented for EquiX. Intuitively, a query is an "example" of the documents that should appear in the output. By formulating an EquiX query the user can specify documents that she would like to find. She can specify constraints on the data that should appear in the documents. We call such constraints *content constraints*. She can also specify constraints on the meta-data, or structure, of the documents. We call such constraints *structural constraints*. In addition, the user can specify *quantification constraints* which constrain the data and meta-data that should appear in the resulting documents by determining how the content and structural constraints should be applied to a document.

The user formulates her query interactively. The user chooses a catalog (d, S). Only documents in S will be searched (queried). At first a *minimal query* is displayed. In a minimal query, only the root element name of d is displayed. A minimal query looks similar to an empty form for querying using a search engine (see Figure 3). The user can then add content constraints by filling in the form, or add structural constraints by expanding elements that are displayed. When an element is expanded, its attributes and subelements, as defined in d, are displayed. The user can add content constraints to the elements and attributes. The user can also specify the quantification that should be applied to each element and attribute, i.e., quantification constraints. This can be one of *exists*, *not exists*, *for all*, and *not for all* (written in a user friendly fashion). In addition, the user can choose which elements in the query should appear in the output.

In Figure 4 an expanded concrete query is depicted. This query was formulated by exploring the DTD presented in Section 2. It retrieves the title and description of Wild West movies in

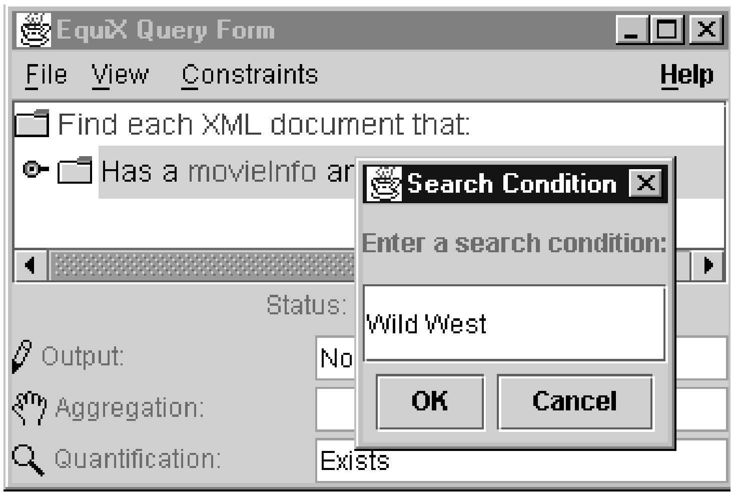

Figure 3: Minimal query that finds documents containing the phrase "Wild West"

which Redford does not star as a villain. Intuitively, answering this query is a two part process:

1. *Search* for Wild West movies. The phrase "Wild West" may appear anywhere in the description of a movie. For example, it may appear in the title or in the movie description. Intuitively, this is similar to a search in a search engine.

2. *Query* the movies to find those in which Redford does not play as a villain. This condition is rather exact. It specifies exactly where the phrases should appear and it contains a quantification constraint. Thus, conceptually, this is similar to a traditional database query.

3.2 Abstract Query Syntax

We present an abstract syntax for EquiX and show how a concrete query is translated to an abstract query.

A boolean function that associates each sequence of alpha-numeric symbols with a truth value among $\{\bot, \top\}$ is a *string matching function*. We assume that there is an infinite set $\mathcal{C}$ of string matching functions, that $\mathcal{C}$ is closed under complement and that the function $\top$ is a member of $\mathcal{C}$. We also assume that each function in C is computable in polynomial time. One such function might be:

$$c_{wild \wedge west}(s) = \begin{cases} \top & \text{if } s \text{ contains the words "wild" and "west"} \\ \bot & \text{otherwise} \end{cases}$$

We define an abstract query below.

Definition 3.1 (Abstract Query) *An abstract query is a rooted directed tree T augmented by four constraining functions and an output set, denoted $Q = (T, l, c, o, q, O)$ where*

- $l : N \to \mathcal{L}$ *is a* labeling function *that associates each node with a label;*

- $c : N \to \mathcal{C}$ *is a* content function *that associates each node with a string matching function;*

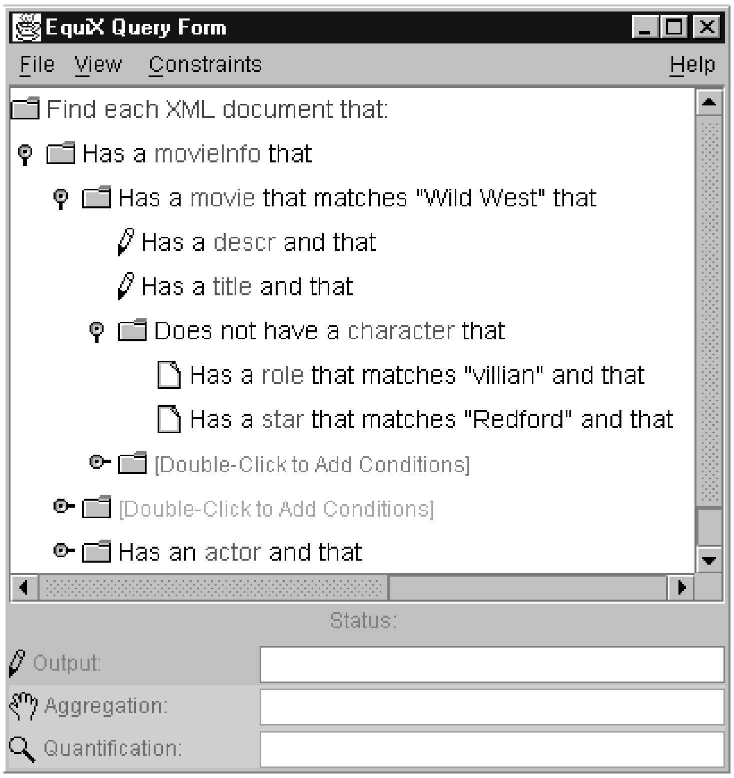

Figure 4: Query that finds titles and descriptions of movies in which Redford isn't a villain

- $o : N \to \{\wedge, \vee\}$ *is an* operator function *that associates each node with a logical operator;*
- $q : E \to \{\exists, \forall\}$ *is a* quantification function *that associates each edge with a quantifier;*
- $O \subseteq N$ *is the set of* projected nodes, *i.e., nodes that should appear in the result.*

Consider a node n. If $o(n) =$ "$\wedge$", we will say that n is an *and-node*. Otherwise we will say that n is an *or-node*. Similarly, consider an edge e. If $q(e) =$ "$\exists$", we will say that e is an *existential-edge*. Otherwise, e is a *universal-edge*.

We give an intuitive explanation of the meaning of an abstract query. The formal semantics is presented in Section 4. When evaluating a query, we will attempt to *match* nodes in a document to nodes in the query. In order for a document node n_X to match a query node n_Q, the function $c(n_Q)$ should hold on the data below n_X. In addition, if n_Q is an and-node (or-node), we require that each (at least one) child of n_Q be matched to a child of n_X. If n_X is matched to n_Q then a child n'_X of n_X can be matched to a child n'_Q of n_Q, only if the edge (n_Q, n'_Q) can be *satisfied* w.r.t. n_X. Roughly speaking, in order for a universal-edge (existential-edge) to be satisfied w.r.t. n_X, all children (at least one child) of n_X that have the same label as n'_Q must be matched to n'_Q.

Note that in a concrete query the user can use the quantifiers "$\exists$", "$\forall$", "$\neg\exists$", "$\neg\forall$" and all nodes are implicitly and-nodes. In an abstract query only the quantifiers "$\exists$", "$\forall$" may be used and the nodes may be either and-nodes or or-nodes. When creating a user interface for our language we found that the concrete query language was generally more intuitive for the user. We present the abstract query language to simplify the discussion of the semantics and query evaluation. Note that the two languages are equivalent in their expressive power.

We address the problem of translating a concrete query to an abstract query. Most of this process is straightforward. The tree structure of the abstract query is determined by the structure of the concrete query. The labeling function l is determined by the labels (i.e., element and attribute names) appearing in the concrete query. The set O is determined by the nodes marked for output by the user.

Translating the quantification constraints is slightly more complicated. As a first step we augment each edge in the query with the appropriate quantifier as determined by the user. We associate each node with the "$\wedge$"-operator and with the content constraint specified by the user. Note that an empty content constraint in a concrete query corresponds to the boolean function $\top$. Next, we propagate the negation in the query. When negation is propagated through an and-node (or-node), the node becomes an or-node (and-node), and the string matching function associated with the node is replaced by its complement. Similarly, when negation is propagated through an existential-edge (universal-edge), the edge becomes a universal-edge (existential-edge). In this fashion, we derive a tree in which each edge is associated with "$\exists$" or "$\forall$" and each node is associated with "$\wedge$" or "$\vee$". The functions o, q, and c are determined by the process described above.

The concrete query in Figure 4 is represented by the abstract query in Figure 5. The string matching functions are specified in italics next to the corresponding nodes. Black nodes are output nodes. In the sequel, unless otherwise specified, the term *query* will refer to an abstract query.

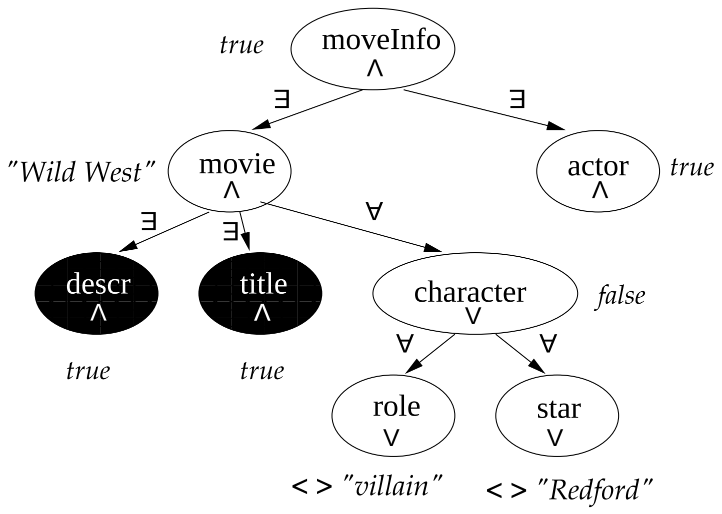

Figure 5: Abstract query for the concrete query in Figure 4. Output nodes are colored black.

Recall the search language requirements we presented in Section 1. We postulated that in a search language, it should not be necessary for the user to specify the format of the result (Criterion 1). In EquiX, by defining the set O, the user only specifies what information she wants the result to include, and does not explicitly detail the format in which it should appear. We suggested that it is important for there to be pattern matching, quantification, and logical expressions for constraining data and meta-data (Criterion 2, 3, and 4). For data, these can all be specified using the content function c. For meta-data, the pattern to which the structure should be matched is specified by T and l, the quantification is specified by q, and logical operators can be specified using o. The result of an EquiX query is a set of XML documents. In Section 6 we show how a DTD for the result documents can be computed. Thus, requerying of results is possible in EquiX (Criterion 5). In Section 5 we show that EquiX queries can be evaluated in polynomial time, and thus, EquiX meets Criterion 6.

4 Search Query Semantics

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 define when a node in a document matches a node in a query. Consider a document X, and a query Q. Suppose that the labeling function of X is l_X and the labeling function of Q is l_Q. We say that a node n_X in X *matches* a node n_Q in Q if $l_X(n_X) = l_Q(n_Q)$. We denote the parent of a node n by $p(n)$. We now define a matching of a document to a query.

Definition 4.1 (Matching) *Let $X = (T_X, l_X)$ be an XML document, with nodes N_X and root r_X. Let $Q = (T_Q, l_Q, c, o, q, O)$ be a query tree with nodes N_Q and root r_Q. A* matching *of X to Q is a function $\mu : N_Q \to 2^{N_X}$, such that the following hold*

1. **Root Matching:** $\mu(r_Q) = \{r_X\}$;
2. **Node Matching:** *if $n_X \in \mu(n_Q)$, n_X matches n_Q;*
3. **Connectivity:** *if $n_X \in \mu(n_Q)$ and n_X is not the root of X, then $p(n_X) \in \mu(p(n_Q))$.*

Note that Condition 1 requires that the root of the document is matched to the root of the query, Condition 2 ensures that matching nodes have the same label, and Condition 3 requires matchings to have a tree-like structure.

We define when a matching of a document to a query is satisfying. We first present some auxiliary definitions. Consider an XML document $X = (T_X, l_X)$, where $T_X = (N_X, E_X, r_X)$. Consider a node n_X in T_X. We differentiate between the *textual content* (i.e., data) contained below the node n_X, and the structural content (i.e., meta-data). When defining the textual content of a node, we take `ID` and `IDREF` values into consideration. We say that n'_X is a *child* of n_X if $(n_X, n'_X) \in E_X$. We say that n'_X is an *indirect child* of n_X if n_X has an attribute of type `IDREF` with the same value as an attribute of type `ID` of n'_X. We denote the textual content of a node n_X as $t(n_X)$, defined as follows:

- If n_X is an atomic node, then $t(n_X) = l_X(n_X)$;
- Otherwise, $t(n_X)$ is the concatenation[2] of the content of its children and indirect children.

We demonstrate the textual content of a node with an example. Recall the XML document depicted in Figure 2. The textual content of Node 9, is "villain 436 Jack Redford". Note that the $t(24)$ includes the value "Jack Redford" since Node 5 is an indirect child of Node 24.

We discuss when a quantification constraint is satisfied. Consider a document X, a query Q and a matching μ of X to Q. Let n_X be a node in X and let $e = (n_Q, n'_Q)$ be an edge in Q. We say n_X *satisfies e with respect to μ* if the following holds

[2] Note that an XML document may be cyclic as a result of ID and IDREF attributes. We take a finite concatenation by taking each child into account only once. In addition, the order in which the concatenation is taken and the ability to differentiate between data that originated in different nodes may affect the satisfiability of a string matching function. This is a technical problem that is taken into consideration in the implementation, by adding an auxillary dividing symbol to the data. We will not elaborate on this point any further.

- If e is an existential-edge then there is a child n'_X of n_X such that n'_X matches n'_Q and $n'_X \in \mu(n'_Q)$.
- If e is a universal-edge then for all children n'_X of n_X, if n'_X matches n'_Q, then $n'_X \in \mu(n'_Q)$.

We define a satisfying matching of a document to a query.

Definition 4.2 (Satisfying Matching) *Let $X = (T_X, l_X)$ be an XML document, and let $Q = (T_Q, l_Q, c, o, q, O)$ be a query tree. Let μ be a matching of X to Q. We say that μ is a* satisfying matching *of X to Q if for all nodes n_Q in Q and for all nodes $n_X \in \mu(n_Q)$ the following conditions hold*

1. *if n_Q is a leaf then $c(n_Q)(t(n_X)) = \top$, i.e., n_X satisfies the string matching condition of n_Q;*
2. *otherwise (n_Q is not a leaf):*
   - (a) *if n_Q is an or-node then n_X satisfies either $c(n_Q)$ or at least one edge incident from n_Q with respect to μ;*
   - (b) *if n_Q is an and-node then n_X satisfies both $c(n_Q)$ and all edges that are incident from n_Q with respect to μ.*

Condition 1 implies that the leaves satisfy the content constraints in Q. Conditions 2a and 2b imply that X satisfies the quantification constraints in Q. The structural constraints are satisfied by the existence of a matching.

Example 4.3 Recall the query in Figure 5 and the document in Figure 2. Two of the satisfying matchings of the document to the query are specified in the following table. There are additional matchings not shown here.

Query Node	μ_1	μ_2
movieInfo	$\{0\}$	$\{0\}$
movie	$\{2\}$	$\{3\}$
descr	$\{10\}$	$\{14\}$
title	$\{11\}$	$\{15\}$
character	$\{12, 13\}$	$\{16\}$
role	$\{25, 27\}$	$\{29\}$
star	$\{26, 28\}$	$\{30\}$
actor	$\{4\}$	$\{5\}$

Note that there is no satisfying matching that matches Node 1 to the movie node in the query because the universal quantification on the edge connecting movie and character cannot be satisfied.

We presented several satisfying matchings of a document to a query. Let μ and μ' be matchings of a document X to a query Q. We define the *union* of μ and μ' in the obvious way. Formally, given a query node n_Q,

$$(\mu \cup \mu')(n_Q) := \mu(n_Q) \cup \mu'(n_Q)$$

There may be an exponential number of satisfying matchings of a given document to a given query. Note, however, that the following proposition holds.

Proposition 4.4 (Union of Matchings) *Let X be an XML document and let Q be a query. Let $\mathcal{M}$ be the set of all satisfying matchings of X to Q. Then the union of all the satisfying matchings in $\mathcal{M}$ is a satisfying matching. Formally,*

$$(\bigcup_{\mu \in \mathcal{M}} \mu) \in \mathcal{M}$$

Proof. It is sufficient to show that the union of any two satisfying matchings μ_1, μ_2 is a satisfying matching. Let $\mu := \mu_1 \cup \mu_2$. Suppose that $X = (T_X, l_X)$ with root r_X and $Q = (T_Q, l_Q, c, o, q, O)$ with root r_Q.

We first show that μ is a matching. It is easy to see that the root of the document is matched to the root of the label since $\mu(r_Q) = \mu_1(r_Q) \cup \mu_2(r_Q) = \{r_X\} \cup \{r_X\} = \{r_X\}$. Now consider $n_X \in \mu(n_Q)$ for some document node n_X and query node n_Q. Then $n_X \in \mu_1(n_Q)$ or $n_X \in \mu_2(n_Q)$. In either case it follows that n_X matches n_Q. Similarly, if n_X is not the root of X then it easily follows that $p(n_X) \in \mu(p(n_Q))$. Thus, μ is a matching.

In a similar fashion it is easy to see that μ is a satisfying matching. This follows since satisfiability is checked for each node separately and thus satisfiability of μ follows directly from satisfiability of μ_1 and μ_2. □

We say that a document X *satisfies* a query Q if there exists a satisfying matching μ of X to Q. We now specify the output of evaluating a query on a single XML document. The result of a query is the set of documents derived by evaluating the query on each document in the queried catalog (i.e., each document that matched the DTD from which the query was derived).

Intuitively, the result of evaluating a query on a document is a subtree of the document (as required in Criterion 1). The subtree contains nodes of three types. Document nodes corresponding to *output* query nodes appear in the resulting subtree. In addition, we include *ancestors* and *descendants* of these nodes. The ancestors ensure that the result has a tree-like structure and that it is a projection of the original document. Recall that the textual content of the document is contained in the atomic nodes of the document tree. Hence, the result must include the descendants to insure that the the textual content is returned.

For a given document, query processing can be viewed as the process of singling out the nodes of the document tree that will be part of the output. Consider a document $X = (T_X, l_X)$ with $T_X = (N_X, E_X, r_X)$ and a query Q with projected nodes O. Let $\mathcal{M}$ be the set of satisfying matchings of X to Q. The output of evaluating the query Q on the document X is the the document defined by projecting N_X on the set $N_R := N_{\text{out}} \cup N_{\text{anc}} \cup N_{\text{desc}}$ defined as

- $N_{\text{out}} := \{n_X \in N_X \mid (\exists n_O \in O)(\exists \mu \in \mathcal{M})\ n_X \in \mu(n_O)\}$, i.e., nodes in X corresponding to projected nodes in Q;
- $N_{\text{anc}} := \{n_X \in N_X \mid (\exists n'_X \in N_{\text{out}})\ n_X$ is an ancestor of $n'_X\}$, i.e., ancestors of nodes in N_{out};
- $N_{\text{desc}} := \{n_X \in N_X \mid (\exists n'_X \in N_{\text{out}})\ n_X$ is an descendant of $n'_X\}$, i.e., descendants of nodes in N_{out}.

We call N_R the *output set* of X with respect to Q.

The result of applying the query in Figure 5 to the document in Figure 2 is depicted in Figure 6. Note that the values of "descr" and "title" are grouped by "movie". This follows naturally from the structure of the original document.

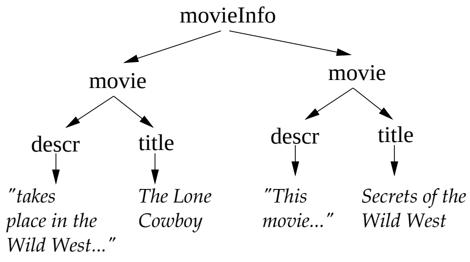

Figure 6: Result Document

5 Query Evaluation

Recall that a query is defined by choosing a catalog and exploring its DTD. Consider a query Q generated from a DTD d in the catalog (d, S). The result of evaluating Q on the database is the set of documents generated by evaluating Q on each document in S.

We present an algorithm for evaluating a query on a document. There may be an exponential number of matchings of a query to a document. Concrete queries contain both quantification and negation. This would appear to be another source of complexity. Thus, it would seem that computing the output of a query on a document should be computationally expensive. 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 [CLR90], 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). Thus, EquiX meets the search language requirement of having polynomial evaluation time (Criterion 6).

In Figure 7 we present a polynomial procedure that computes the output of a document, given a query. Given a document X and query Q, the procedure Query_Evaluate computes the output set N_R of X w.r.t. Q. Note that the value of $t(n_X)$ for each document node n_X can be computed in a preprocessing step in polynomial time. Query_Evaluate uses the procedure Matches shown in Figure 8. Given a query node n_Q and a document node n_X, the procedure Matches checks if it is possible that $n_X \in \mu(n_Q)$ for some matching μ, based on the subtrees of n_Q and n_X.

Note that *path*(n) is the sequence of element names on the path from the root of the query to n, and *anc*(n) (*desc*(n)) is the set of ancestors (descendants) of n. Note also that N_Q are the query nodes and N_X are the document nodes. We use $|N_Q|$ and $|N_X|$ to denote the size of the query and document nodes, respectively. The array *match_array* is an array of size $|N_Q| \times |N_X|$ of boolean values. Observe that in Figure 7 we order the nodes by descending depth. This ensures that when Matches(n_Q,n_X,*match_array*) is called, the array *match_array* is already updated for all the children of n_Q and n_X. The procedure Query_Evaluate does not explicitly create any matchings. However, the following theorem holds.

Theorem 5.1 (Correctness of Query_Evaluate) *Given a document X and a query Q, the algorithm* Query_Evaluate *computes the output set of X w.r.t. Q.*

In Appendix A we prove this theorem. It can be shown that the procedure Query_Evaluate runs in polynomial time in combined complexity. Let $|D|$ be the size of the data in document X, i.e., the size of X when ignoring X's meta-data. Formally, $|D| = |t(r_X)|$. Let $C(m)$ be an upper-bound on the runtime of computing a string-matching constraint on a string of size m. Recall that $C(m)$ is polynomial in m.

Algorithm Query_Evaluate
Input Document $X = (T_X, l_X)$ s.t. $T_X = (N_X, E_X, r_X)$,
Query tree $Q = (T_Q, l_Q, c, o, q, O)$ s.t. $T_Q = (N_Q, E_Q, r_Q)$
Output $N_R \subseteq N_X$, i.e., the outputed document nodes

Initialize *match_array*[][] to **false**
$Queue_1 := N_Q$, ordered by descending depth
While (**not** isEmpty($Queue_1$)) **do**
  $n_Q :=$Dequeue($Queue_1$)
  For all $n_X \in N_X$ such that $path(n_X) = path(n_Q)$ **do**
    $match\_array[n_Q][n_X]:=$ Matches(n_Q, n_X,$match\_array$)
$N_R := \emptyset$
$Queue_2 := N_Q$, ordered by ascending depth
While (**not** isEmpty($Queue_2$)) **do**
  $n_Q :=$Dequeue($Queue_2$)
  For all $n_X \in N_X$ **do**
    If ($n_Q \neq r_Q$ and **not** $match\_array[p(n_Q)][p(n_X)]$) **then**
      $match\_array[n_Q][n_X]$:= **false**
    Else If ($match\_array[n_Q][n_X]$ **and** $n_Q \in O$) **then**
      $N_R := N_R \cup \{n_X\} \cup anc(n_X) \cup desc(n_X)$
Return N_R

Figure 7: Evaluation of an EquiX Query

Theorem 5.2 (Polynomial Complexity) *Given document X and a query Q, the algorithm* Query_Evaluate *runs in time $O(|N_X| \cdot |N_Q| \cdot (|N_Q| \cdot |N_X| + C(|D|)))$.*

Proof. The initialization stage, i.e., the sorting of $Queue_1$ can be done in $O(|N_Q|lg(|N_Q|))$. The first "while" loop runs $O(|N_X||N_Q|)$ times and in each iteration calls the procedure Matches which runs in time $O(|N_Q||N_X| + C(|D|))$. Once again, initialization of $Queue_2$ can be done in $O(|N_Q|lg(|N_Q|))$. The second while loop runs in time $O(|N_X|^2|N_Q|)$. Therefore, the algorithm Query_Evaluate runs in time $O(2|N_Q|lg(|N_Q|) + |N_X||N_Q|(|N_Q||N_X| + C(|D|)) + |N_X|^2|N_Q|)$, which is equal to $O(|N_X||N_Q|(|N_Q||N_X| + C(|D|)))$ as required. □

6 Creating a Result DTD

In Section 5 we described the process of evaluating a query on a database. Query evaluation generates a set of documents. A query is formed using a chosen DTD, called the *originating DTD*, and only documents strictly conforming to the originating DTD will be queried. Thus, in order to allow *iterative querying* or *requerying of results*, a DTD for the resulting documents must be defined. Given a query Q, if any possible result document must conform to the DTD d_R, we say that d_R is a *result DTD for* Q. In this section we present a procedure that given a query Q, computes in polynomial time a result DTD for Q. Thus, we show that EquiX fulfills the search language requirement of ability to perform requerying (Criterion 5).

Procedure	$\mathsf{Matches}(n_Q, n_X, match\_array)$
Input	A query node n_Q
	A document node n_X
	An array $match\_array$
Output	**true** if n_X may be in $\mu(n_Q)$ for a matching μ,
	based on the subtrees of n_X and n_Q, and **false** otherwise

$tc := c(n_Q)(t(n_X))$
If n_Q is a terminal node **return** tc
Let M_Q be the set of children of n_Q in Q
For each $m_Q \in M_Q$ do:
  Let M_X be the set of children, m_X, of n_X in X such that $l_X(m_X) = l_Q(m_Q)$
  If (n_Q, m_Q) is an existential-edge **then**
    $status(m_Q) := \bigvee_{m_X \in M_X} match\_array[m_Q][m_X]$
  Else $status(m_Q) := \bigwedge_{m_X \in M_X} match\_array[m_Q][m_X]$
If n_Q is an or-node **then**
  return $tc \vee (\bigvee_{m_Q \in M_Q} status(m_Q))$
Else **return** $tc \wedge (\bigwedge_{m_Q \in M_Q} status(m_Q))$

Figure 8: Satisfaction of a Node Procedure

A DTD is a set of *element definitions*, and *attribute list definitions*. An element definition has the form `<!ELEMENT` e φ`>`, where e is the element name being defined and φ is its *content definition*. An attribute list definition has the form `<!ATTLIST` e $\psi_1 \dots \psi_n$`>`, where e is an element name and $\psi_1 \dots \psi_n$ are definitions of attributes for e. The set of element names defined in a DTD d is its *element name set*, denoted $\mathcal{E}_d$.

Consider a query $Q = (T, l, c, o, q, O)$ formulated from a DTD d. We say that element name e' is a *descendant* of element name e in d if e' may be nested within an element e in a document conforming to d. Formally, e' is a descendant of e if

- e' appears in the content definition of e *or*
- e' is a descendant of an element name e'' which appears in the content definition of e.

We say that e is an *ancestor* of e' in d if e' is a descendant of e in d. Note that the element name e may appear in a document resulting from evaluating Q if there is a node $n_O \in O$ such that $l(n_O) = e$. Additionally, e may appear in the output if e is an ancestor or descendant in d of an element e' that meets the condition presented in the previous sentence. Thus, given a query, we can compute in linear time the element name set $\mathcal{E}_{d_R}$ of the result DTD d_R.

In order to compute the result DTD of a query Q, we must compute the content definitions and attribute list definitions for the elements in $\mathcal{E}_{d_R}$. In the result DTD, we take the attribute list definitions for the elements in $\mathcal{E}_{d_R}$ as defined in the originating DTD but change all attributes to be of type `#IMPLIED`. Note that the root element name r of the originating DTD will always be in $\mathcal{E}_{d_R}$. It is easy to see that r is also the designated root element name of d_R.

In Figure 9 we present an algorithm that computes the content definition for an element in $\mathcal{E}_{d_R}$. Intuitively, any elements that will not appear in the result of a query must be removed from the original DTD in order to form the result DTD. In addition, elements will only appear in result documents if query constraints are satisfied. Thus, this possible appearance of elements may be taken into account when formulating d_R. The algorithm Create_Content_Definition uses the procedure presented in Figure 10 in order to simplify the content definition it creates. The result DTD is created by computing the content definitions for all $e \in \mathcal{E}_{d_R}$ and adding the attribute list definitions. Note that in the algorithm, $dtd\_desc(e', e)$ is true if e is a descendant of e' in the DTD D. In addition, $anc(n_X, O)$ is true if n_X is an ancestor of some node in O.

Algorithm Create_Content_Definition
Input An element $e \in \mathcal{E}_{d_R}$
A query Q with nodes N_Q, edges E_Q and projected nodes O
The originating DTD d with content definition φ_e for e
Output The content definition of e in the result DTD

If $(\exists n_O \in O)$ s.t. $((l(n_O) = e)$ or
$(l(n_O) = e'$ and $dtd\_desc(e', e)))$ **then**
$\varphi := \varphi_e$
Else $\varphi := \emptyset$
For each $(n_Q \in N_Q)$ s.t. $((l(n_Q) = e)$ and $(anc(n_Q, O)))$ **do**
$\varphi' := \varphi_e$
For all elements e' in φ_e **do**
If $(\exists n'_Q \in N_Q)$ s.t. $path(n_Q) = path(n'_Q)$ and
$(\exists n''_Q \in N_Q)$ s.t. $(n'_Q, n''_Q) \in E_Q$ and $l(n''_Q) = e'$ and $(anc(n''_Q, O)))$ **then**
Replace all occurrences of e' in φ' with $(e'?)$
Else Replace all occurrences of e' in φ' with $\emptyset$
$\varphi := \varphi \mid \varphi'$
Return Simplify(φ).

Figure 9: Content Definition Generation Algorithm

Theorem 6.1 (Correctness of DTD Creation) *Let Q be a query with an originating DTD d and let X be a document. Suppose that the result of evaluating Q on X is the XML document R. Then R strictly conforms to the result DTD as formed by the process described above. In addition, the computation of the result DTD can be performed in time $O(|d||Q|)$.*

Proof. We first prove correctness. Consider a specific occurence of an document node n_R with with an element name of e appearing in a result document. Clearly, an element with name e can appear in a result document only if $e \in \mathcal{E}_{d_R}$. Thus, e has a content definition in the result DTD. The content definition of e is a disjunction of content definitions. It is sufficient to show that one of the definitions in the disjunction is satisfied with respect to the children of n_R. There are three possible causes for this occurence of n_R in the result document:

1. There is a matching μ such that $n_R \in \mu(n_Q)$ for some output node n_Q in the query. Thus, there is an output node n_Q in the query with label e. Therefore, according to the algorithm, we take the original definition of e as one of the disjuncts of the new definition of e. Note that in this case, all of n_R's children will appear in the result. Thus, the children of n_R satisfy the definition of e in the result DTD.

Procedure $\mathsf{Simplify}(\varphi)$
Input Content definition φ
Output Simplified content definition of φ

While there is a change in φ
  Apply the following rules to φ or its subexpressions

1. $(t, \emptyset)$	$\Rightarrow$	t	5. $(\emptyset)?$	$\Rightarrow$	$\emptyset$
2. $(\emptyset, t)$	$\Rightarrow$	t	6. $(\emptyset)*$	$\Rightarrow$	$\emptyset$
3. $(t \mid \emptyset)$	$\Rightarrow$	$t?$	7. $(\emptyset)+$	$\Rightarrow$	$\emptyset$
4. $(\emptyset \mid t)$	$\Rightarrow$	$t?$			

If $\varphi = \emptyset$ **then Return** `EMPTY`
Else Return φ

Figure 10: Definition Simplifying Algorithm

2. The node n_R is a descendant of a node matched to an output query node and Case 1 does not hold. This case can be proved in the same manner as the previous case.

3. The node n_R is matched to a query node n_Q that is an ancestor of an output node and Cases 1 and 2 do not hold. Note that it is possible that some of n_R's children in the document do not appear in the result. Specifically, a child of n_R with element name e' cannot appear in the result if there is no query node n'_Q with the same path from root as n_Q and with a child labeled with e' that is an ancestor of an output node. (This easily follows from the definition of the output of a query.) In the content definition that we create for e according to n_Q these elements are replaced by the empty element since they cannot appear in the output. All other elements are made optional by the addition of the "?" symbol. Thus, clearly the content definition defined according to n_Q will be satisfied by the children of n_R.

Thus, the algorithm is correct.

The algorithm Create_Content_Definition is called at most $|d|$, each time for a different element name e. The algorithm Create_Content_Definition then goes over the nodes in the query with label e. For each such node, a content definition is created which is of size $O(|d|)$. Thus, when amortizing the cost of the creation over all the query nodes, we derive that the result DTD can be created in time $O(|d||Q| + |d|) = O(|d||Q|)$. □

Note that it follows from Theorem 6.1 that the result DTD is polynomial in the size of the original DTD and the query. The compactness of the result DTD makes the requerying process simpler, since requerying entails exploring the result DTD.

According to Theorem 6.1, the resulting documents conform to the result DTD. The question arises as to how precisely the result DTD describes the resulting documents. In order to answer this question we define a partial order on DTDs [PV99]. Given a DTD d we denote the set of XML documents that strictly conform to d as $conf(d)$. Given DTDs d and d' we say that d is *tighter* than d', denoted $d \preceq d'$, if $conf(d) \subseteq conf(d')$. We say that d is *strictly tighter* that d', denoted $d \prec d'$, if $conf(d) \subset conf(d')$.

Intuitively, it would be desirable to find a result DTD d_R that is as tight as possible, under the restriction that all possible result documents must strictly conform to d_R. However, our algorithm does not necessarily find the tightest possible result DTD. In other words, our algorithm may create a result DTD d_R although there exists a DTD d'_R to which all resulting documents must strictly conform and $d'_R \prec d_R$. If d_R is the tightest possible result DTD, we call d_R a *minimal result DTD*. A minimal result DTD may not be unique. For a comprehensive discussion, see [PV99]. According to the following Proposition, there is a query and DTD for which a minimal result DTD must be exponential in the size of the original DTD.

Proposition 6.2 (Exponential Result DTD) *There is a query Q created from an originating DTD d, such that if d' is a minimal result DTD of Q, then d' is of size $\mathcal{O}(\,|d|!\,)$.*

Proof. Consider a DTD d with the root element r. Suppose that d contains the element definition

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

Let $Q = (T_Q, l_Q, c, o, q, O)$ be a query with with root n_r and children $n_1, \ldots, n_k$. Suppose that $l(n_r) = r$ and $l(n_i) = a_i$ for all i. We assume that all the nodes are and-nodes and all the edges are existential-edges. Suppose in addition that $O = \{n_1, \ldots, n_k\}$ and c maps each node to an arbitrary condition. We can conclude that each of the element names $a_1, \ldots, a_k$ will appear at least once in any result document. However, these element names they can appear in any order. Thus, a minimal result DTD must consider $k!$ different orderings of the elements, proving the claim.[3] $\square$

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 requery previous results. Thus, our algorithm for creating result DTDs actually returns a convenient DTD, although it is not always minimal.

7 Extending EquiX Queries

EquiX can be extended in many ways to yield a more powerful language. In this section we present two extensions to the EquiX language. These extensions add additional querying ability to EquiX. After extending EquiX, the search language requirements 1 through 6 are still met. However, it is a matter of opinion if EquiX still fulfills Criterion 7 requiring simplicity of use. Thus, these extensions are perhaps more suitable for expert users.

7.1 Adding Aggregation Functions and Constraints

We extend the EquiX language to allow computing of aggregation functions and verification of aggregation constraints. We call the new language EquiX$^{\text{agg}}$.

We extend the abstract query formalism to allow aggregation. An *aggregation function* is one of $\{count, min, max, sum, avg\}$. An *atomic aggregation constraint* has the form $f\theta v$ where f is an aggregation function, $\theta \in \{<, \leq, =, \neq, \geq, >\}$, and v is a constant value. An *aggregation constraint* is a (possibly empty) conjunction of atomic aggregation constraints. In EquiX$^{\text{agg}}$ a query is a tuple $(T, l, c, o, q, a, a_c, O)$ as in EquiX, augmented with the following functions:

- a is an *aggregation specifying function* that associates each node with a (possibly empty) set of aggregation functions;

[3]The reader should recall that the order of the document nodes defines the order of the result document nodes, while the order of the query nodes has no influence on the result. The need for an exponential size DTD hinges on this fact.

- a_c is an *aggregation constraint function* that associates each node with an aggregation constraint.

Given a node n_Q, the aggregation specifying functions $a(n_Q)$ (and similarly the aggregation constraint functions) are applied to $t(n_X)$ for all $n_X \in \mu(n_Q)$. Note that the *min* and *max* functions can only be applied to an argument whose domain is ordered. Similarly, the aggregation functions *sum* and *avg* can only be applied to sets of numbers. There is no way to enforce such typing using a DTD (although it is possible using an XML Schema [Con]). When a function is applied to an argument that does not meet its requirement, its result is undefined.

The function a adds the computed aggregation values to the output. This is similar to placing an aggregation function in the SELECT clause of an SQL query. The function a_c is used to further constrain a query. This is similar to the HAVING clause of an SQL query.

In order to use an aggregation function in an SQL query, one must include a GROUP-BY clause. This clause specifies on which variables the grouping should be performed, when computing the result. To simplify the querying process, we do not require the user to specify the GROUP-BY variables. EquiX$^{\text{agg}}$ uses a simple heuristic rule to determine the grouping variables. Suppose that $a(n_Q) \neq \emptyset$ for some node n_Q. Let n_O be the lowest node above n_Q in Q for which one of the following conditions hold

- $n_O \in O$ *or*
- n_O is an ancestor of a node in O.

Note that n_O is the lowest node above n_Q where both textual content and aggregate values may be combined. EquiX$^{\text{agg}}$ groups by n_O when computing the aggregation functions on n_Q. In a similar fashion, EquiX$^{\text{agg}}$ performs grouping in order to compute aggregation constraints.

Our choice for grouping variables is natural since it takes advantage of the tree structure of the query, and thus, suggests a polynomial evaluation algorithm. It is easy to see that adding aggregation functions and constraints does not affect the polynomiality of the evaluation algorithm. The algorithm for creating a result DTD must also be slightly adapted in order to take into consideration the aggregation values that are retrieved. Thus, EquiX$^{\text{agg}}$ meets the search language requirements 1 through 6.

7.2 Querying Ontologies using Regular Expressions

In EquiX, the user chooses a catalog and queries only documents in the chosen catalog. It is possible that the user would like to query documents conforming to different DTDs, but containing information about the same subject. In EquiX$^{\text{reg}}$ this ability is given to the user. Thus, EquiX$^{\text{reg}}$ is useful for information integration.

An ontology, denoted $\mathcal{O}$, is a set of terms whose meanings are well known. Note that an ontology can be implemented using XML Namespaces [BHL99]. We say that a document X *can be described by* $\mathcal{O}$ if some of the element and attribute names in X appear in $\mathcal{O}$. When formulating a query, the user chooses an ontology of terms that describes the subject matter that she is interested in querying. Documents that can be described by the chosen ontology will be queried. A query tree in EquiX$^{\text{reg}}$ is a tuple (T, l, c, o, q, O) as in EquiX. However, in EquiX$^{\text{reg}}$, l is a function from the set of nodes to $\mathcal{O}$.

Semanticly, an EquiX$^{\text{reg}}$ query is interpreted in a different fashion from an EquiX query. Each edge is implicitly labeled with the "+" symbol. Intuitively, an edge in a query corresponds to a sequence of one or more edges in a document. We adapt the definition of satisfaction of an edge (presented in Section 4) to reflect this change.

Consider a document X, a query Q and a matching μ of X to Q. Let n_X be a node in X and let $e = (n_Q, n'_Q)$ be an edge in Q. We say n_X *satisfies* e *with respect to* μ if the following holds

- If e is an existential-edge then there is a *descendent* n'_X of n_X such that n'_X matches n'_Q and $n'_X \in \mu(n'_Q)$.
- If e is a universal-edge then for all *descendents* n'_X of n_X, if n'_X matches n'_Q, then $n'_X \in \mu(n'_Q)$.

Note that the only change in the definition was to replace the words child and children with descendent and descendents.

In a straightforward fashion, we can modify the query evaluation algorithm to reflect the new semantics presented. The new algorithm remains polynomial under combined complexity. Note that it is no longer possible to create a result DTD if we do not permit a result DTD to contain content definitions of type `ANY`. This results from the possible diversity of the documents being queried. However, $\text{EquiX}^{\text{reg}}$ still meets Criterion 5 (i.e., ability to requery results), since the resulting documents can be described by the chosen ontology. Thus, $\text{EquiX}^{\text{reg}}$ meets the search language requirements 1 through 6.

8 Conclusion

In this paper we presented design criteria for a search language. We defined a specific search language for XML, namely EquiX, that fulfills these requirements. Both a user-friendly concrete syntax and a formal abstract syntax were presented. We defined an evaluation algorithm for EquiX queries that is polynomial even under combined complexity. We also presented a polynomial algorithm that generates a DTD for the result documents of an EquiX query.

We believe that EquiX enables the user to search for information in a repository of XML documents in a simple and intuitive fashion. Thus, our language is especially suitable for use in the context of the Internet. EquiX has the ability to express complex queries with negation, quantification and logical expressions. We have also extended EquiX to allow aggregation and limited regular expressions. To summarize, EquiX is unique in its being both a powerful language and a polynomial language.

Several XML query languages have been proposed recently, such as XML-QL [DFF+98], XQL [RLS98] and Lorel [GMW99]. These languages are powerful in their querying ability. However, they do not fulfill some of our search language requirements. In these languages, the user can perform restructuring of the result. Thus, the format of the result must be specified, in contradiction to Criterion 1. Furthermore, XML-QL and XQL are limited in their ability to express quantification constraints (Criterion 3). Most importantly, none of these languages guarantee polynomial evaluation under combined complexity (Criterion 6).

As future work, we plan to extend the ability of querying ontologies and to allow more complex regular expressions in EquiX. XML documents represent data that may not have a strict schema. In addition, search queries constitute a guess of the content of the desired documents. Thus, we plan to refine EquiX with the ability to deal with incomplete information [KNS99] and with documents that "approximately satisfy" a query. Search engines perform an important service for the user by sorting the results according to their quality. We plan on experimenting to find a metric for ordering results that takes both the data and the meta-data into consideration.

As the World-Wide Web grows, it is becoming increasingly difficult for users to find desired information. The addition of meta-data to the Web provides the ability to both search and query the Web. Enabling users to formulate powerful queries in a simple fashion is an interesting and challenging problem.

References

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BHL99] T. Bray, D. Hollander, and A. Layman. Namespaces in XML, 1999. Available at `http://www.w3.org/TR/1999/REC-xml-names-19990114`.

[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`.

[Bra98] N. Bradley. *The XML companion.* Addison-Wesley, 1998.

[BYKK+99] Z. Bar-Yossef, Y. Kanza, Y. Kogan, W. Nutt, and Y. Sagiv. Querying Semantically Tagged Documents on the World-Wide Web. In *Proc. 4th International Workshop on Next Generation Information Technologies and Systems*, Zichron Yaakov (Israel), July 1999. Springer-Verlag.

[CKK+99] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX—Eeasy Quering in XML Databases. In *International Workshop on the Web and Databases (WebDB)*, 1999.

[CKK+00] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. Combining the power of searching and querying. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS)*, volume 1901 of *Lecture Notes in Computer Science*, pages 54–65, Eilat (Israel), September 2000. Springer-Verlag.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[Con] The World-Wide Web Consortium. XML Schema. Available in three parts at `http://www.w3.org/TR/xmlschema-0/`, `http://www.w3.org/TR/xmlschema-1/` and `http://www.w3.org/TR/xmlschema-2/`.

[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/`.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *International Workshop on the Web and Databases (WebDB)*, 1999.

[GW98] R. Goldman and J. Widom. Interactive Query and Search in Semistructured Databases. In *International Workshop on the Web and Databases*, Valencia, Spain, 1998.

[KNS99] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with Incomplete Answers over Semistructured Data. In *Proc. 18th Symposium on Principles of Database Systems*, pages 227–236, Philadelphia (Pennsylvania, USA), June 1999. ACM Press.

[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`.

A Correctness of Query_Evaluate

In this Section we prove the correctness of the algorithm Query_Evaluate presented in Figure 7. We first prove some necessary lemmas.

Lemma A.1 *Let n_X be a node in a document X and let n_Q be a node in a query Q. If there exists a matching μ of X to Q such that $n_X \in \mu(n_Q)$ then $path(n_X) = path(n_Q)$.*

Proof. Suppose that $n_X \in \mu(n_Q)$. We show by induction on the depth of n_Q that $path(n_X) = path(n_Q)$. Suppose that the labeling function of Q is l_Q and that the labeling function of X is l_X.

Case 1: Suppose that n_Q is the root of Q. According to Condition 2 in Definition 4.1, it holds that $l_Q(n_Q) = l_X(n_X)$. According to Condition 1 in Definition 4.1, n_X is the root of X. Thus, clearly the claim holds.

Case 2: Suppose that n_Q is of depth m. Once again, according to Condition 2 in Definition 4.1, it holds that $l_Q(n_Q) = l_X(n_X)$. In addition, $p(n_X) \in \mu(p(n_Q))$ (see Condition 3 in Definition 4.1). Note that $p(n_Q)$ is of depth $m-1$. Thus, by the induction hypothesis, $path(p(n_X)) = path(p(n_Q))$. Our claim follows. □

We present an auxiliary definition. We define the *height* of a query node n_Q, denoted $h(n_Q)$, as

$$h(n_Q) = \begin{cases} 0 & \text{if } n_Q \text{ is a leaf node} \\ max\{h(n'_Q) | n'_Q \text{ is a child of } n_Q\} + 1 & \text{otherwise} \end{cases}$$

We show that the algorithm implicitly defines a satisfying matching μ_R. The nodes that are returned are those corresponding to output nodes in μ_R, and their ancestors and descendents.

We define the function $\mu_R : N_Q \to 2^{N_X}$ in the following way:

$$n_X \in \mu_R(n_Q) \iff match\_array[n_Q, n_X] = \text{"true"}$$

Note that we consider the values of *match_array* at the end of the evaluation of Query_Evaluate. We call μ_R the *retrieval function* of Query_Evaluate w.r.t. X and Q.

Lemma A.2 (Retrieval Function is a Satisfying Matching) *Let X be a document, let Q be a query and let μ_R be the retrieval function of* Query_Evaluate *w.r.t. X and Q. Then μ_R is a satisfying matching of X to Q.*

Proof. We show that μ_R is a matching, i.e., that μ_R meets the conditions in Definition 4.1.

- **Roots Match:** The only node that has the same path as r_Q is r_X. Thus, the only time that the function Matches is called for the root of the query is with the root of the document. Thus, the value of $\mu_R(r_Q)$ must either be either $\{r_X\}$ or $\emptyset$. However, it is easy to see that if $\mu_R(r_Q) = \emptyset$ then Query_Evaluate returns $\emptyset$. Thus, it must hold that $\mu_R(r_Q) = \{r_X\}$.
- **Node Matching:** If $n_X \in \mu_R(n_Q)$ then Matches was called with n_Q and n_X. Thus n_X and n_Q have the same path, and hence, n_X matches n_Q.
- **Connectivity:** If $match\_array[p(n_Q), p(n_X)]$ does not hold, then $match\_array[n_Q, n_X]$ is assigned the value "false". Therefore, clearly the connectivity requirement of a matching holds.

We now show that μ_R is a satisfying matching (see Definition 4.2). Suppose that $n_X \in \mu_R(n_Q)$ for a document node n_X and a query node n_Q. Note that this implies that Matches returned the value "true" when applied to n_Q and n_X. We prove by induction on the height of n_Q that the appropriate condition holds. We consider three cases.

- Suppose that $h(n_Q) = 0$. Then n_Q is a leaf and $c(n_Q)(t(n_X))$ must hold as required.
- Suppose that $h(n_Q) > 0$ and that n_Q is an or-node. The procedure Matches returned the value "true" when applied to n_Q and n_X. Therefore, one of the following must hold:

  1. The condition $c(n_Q)(t(n_X)) = \top$ holds.
  2. At the time of application of Matches to n_Q and n_X, there was a child m_Q of n_Q and a child m_X of n_X such that *match_array*[m_Q,m_X] had the value "true". Note that it follows that m_X matches m_Q. The value of *match_array*[m_Q,m_X] will not be changed to "false" during the evaluation of Query_Evaluate since *match_array*[n_Q,n_X] is "true". Thus, $m_X \in \mu_R(m_Q)$.

  In either case Condition 2a from Definition 4.2 holds as required.
- Suppose that $h(n_Q) > 0$ and that n_Q is an and-node. We omit the proof as it is similar to the previous case.

Thus, μ_R is a satisfying matching as required. □

We say that a matching μ *contains* a matching μ' if $\mu(n_Q) \supseteq \mu'(n_Q)$ for all query nodes n_Q. We will show that the retrieval function contains all other satisfying matchings.

Lemma A.3 (Retrieval Function Containment) *Let X be a document, let Q be a query and let μ_R be the retrieval function of* Query_Evaluate *w.r.t. X and Q. Suppose that μ is a satisfying matching of X to Q. Then μ_R contains μ.*

Proof. Suppose that μ is a satisfying matching. We show by induction on the height of n_Q that $\mu(n_Q) \subseteq \mu_R(n_Q)$. We first consider the values assigned to *match_array* during the first pass (the bottom-up pass) of the algorithm.

- Suppose that $h(n_Q) = 0$. Suppose that $n_X \in \mu_R(n_Q)$. Then, according to Lemma A.1 $path(n_X) = path(n_Q)$. Thus, Matches will be called on n_X and n_Q. The condition $c(n_Q)(t(n_X))$ holds. Thus, *match_array*[n_Q,n_X] will be assigned the value "true".
- Suppose that $h(n_Q) > 0$ and n_Q is an or-node. Suppose that $n_X \in \mu_R(n_Q)$. Once again, according to Lemma A.1 $path(n_X) = path(n_Q)$. Thus, Matches will be called on n_X and n_Q. It also follows that one of the following must hold:

  1. The value of $c(n_Q)(t(n_X))$ is "true". Thus, Matches returns true.
  2. There is a child m_Q of n_Q and a child m_X of n_X such that m_X matches m_Q and $m_X \in \mu(m_Q)$. Note that $h(m_Q) < h(n_Q)$. Thus, by the induction hypothesis, the value of *match_array*[m_Q,m_X] after the first pass of the algorithm is "true". Thus Matches returns true when called on n_Q and n_X.
- Suppose that $h(n_Q) > 0$ and n_Q is an and-node. We omit the proof as it is similar to the previous case.

It is easy to see that it follows from the connectivity of μ that if $n_X \in \mu(n_Q)$ then the value of *match_array*[n_Q,n_X] will not be changed to "false". Thus, μ is contained in μ_R as required. □

We can now prove the theorem required.

Theorem A.4 (Correctness of Query_Evaluate) *Given document X and a query Q, the algorithm* Query_Evaluate *retrieves the output set of X w.r.t. Q.*

Proof. Let X be a document and Q be a query. We show that a document node n_X is returned by Query_Evaluate if and only if n_X is in the output set of X w.r.t. Q.

"$\Leftarrow$" Suppose that n_X is in the output set of X w.r.t. Q. Then there is a satisfying matching μ of X to Q such that and an output node n_Q in Q such that either $n_X \in \mu(n_Q)$ or n_X is an ancestor or descendent of a node in $\mu(n_Q)$. Let μ_R be the retrieval function of Query_Evaluate w.r.t. X and Q. According to Lemma A.3 μ is contained in μ_R. Thus, clearly n_X is returned by Query_Evaluate.

"$\Rightarrow$" Suppose that n_X is returned by Query_Evaluate. Let μ_R be the retrieval function of Query_Evaluate w.r.t. X and Q. Then there is an output node n_Q in Q such that either $n_X \in \mu_R(n_Q)$ or n_X is an ancestor or descendent of a node in $\mu_R(n_Q)$. According to Lemma A.2 μ_R is a satisfying matching of X to Q. Thus, n_X is in the output set of X w.r.t. Q. $\square$