Combining the Power of Searching and Querying*

Sara Cohen[1], Yaron Kanza[1], Yakov Kogan[1], Werner Nutt[2], Yehoshua Sagiv[1], and Alexander Serebrenik[3]

[1] Computer Science Dept., The Hebrew University, Jerusalem, Israel
{sarina,yarok,yakov,sagiv}@cs.huji.ac.il
[2] Department of Computing and Electrical Engineering
Heriot-Watt University, Edinburgh, Scotland
nutt@cee.hw.ac.uk
[3] Computer Science Dept., K. U. Leuven, Heverlee, Belgium
Alexander.Serebrenik@cs.kuleuven.ac.be

Abstract. EquiX is a search language for XML that combines the power of querying with the simplicity of searching. Requirements for search languages are discussed and it is shown that EquiX meets the necessary criteria. Both a graphical abstract syntax and a formal concrete syntax are presented for EquiX queries. In addition, the semantics is defined. It is shown that EquiX has an evaluation algorithm that 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. It is difficult to query the meta-data content in such pages using a search engine. Only the data can easily 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 as 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 Web pages. 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 [7], XML-QL [4] and Xmas [6], are much richer than search-engine query languages. However, the limited expressiveness of search engines appears to be an advantage

* This research was supported by grant 9481-1-98 of the Israeli Ministry of Science.

in the context of the Web. Many Internet users would find it hard to formulate SQL queries. In comparison, 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. We believe that the Web gives rise to a new concept in query languages, namely *search languages*. 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. Many times the schema is (partially) unknown when a search is performed. It is virtually impossible to perform many types of searches in traditional database languages, such as SQL. For example, without knowing the schema of a database, it is impossible to find all tuples in all tables that contain the word "Jones". In fact, even if the schema is known, it is quite difficult to formulate such a query. Querying is a precise process in which the user specifies exactly the information she seeking and where it is located in relation to the schema. 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 designing of a search language and processor:

1. **Format of Results:** A search result of a search query should be a set of documents or sections of documents that satisfy the query. When searching, the user is interested in *finding* information. Thus, restructuring of documents to compute results is not necessary.
2. **Pattern Matching, Quantification and Logical Expressions:** Search engines allow pattern matching, quantification (using "+" and "-" symbols, etc.) and logical expressions (using `AND`, `OR` and `NOT`, etc.). Currently, these can be performed on the data only. We propose to extend these capabilities to be performed on meta-data. Pattern matching on the meta-data allows a user to formulate a search query without knowing the exact structure of the document. Allowing quantification and logical expressions on both data and meta-data enriches the query language and makes it more powerful.
3. **Iterative Searching Ability:** The result of a search query may contain hundreds, if not thousands, of documents. Thus, it is important to allow requerying of previous results. This enables users to search for the desired information iteratively, until such information is found.
4. **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*.

5. **Simplicity:** 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 4. 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. In an extension to EquiX we allow aggregation and a limited class of regular expressions ([3]). Both searching and querying can be performed using the EquiX language. EquiX also simplifies the querying process by automatically generating the format of the result and a corresponding DTD.

This paper extends previous work [2]. An extended version of this paper can be found in [3]. 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 the evaluation of EquiX queries is discussed. Section 6 concludes.

2 Data Model

We define a data model for querying XML documents [1]. At first, we assume that each XML document has a given DTD. In [3] this assumption is relaxed. 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 a 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. A directed tree is *rooted* if there is a designated node $r \in N$, such that every node in N is reachable from r in T. We call r the *root* of 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 by use of nesting. We will represent a 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, a 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. However, we can assume, without loss of generality, that all XML documents are rooted trees. An XML document that gives rise to a rooted tree is said to be *rooted* and the element that corresponds to the root of the tree is called the *root element*.

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 1 (XML Document). *An* XML document *is a pair* (T, l) *s.t.*

- $T = (N, E, r)$ *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 [1]) and
2. the function l assigns the label e to the root r (i.e., $l(r) = e$).

The following DTD with root element name `movieInfo` describes information about movies.

```
<!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>
```

In Figure 1 an XML document containing movie information is depicted. This document strictly conforms to the DTD presented above. Note that the nodes in Figure 1 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.

[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 omit this in the paper.

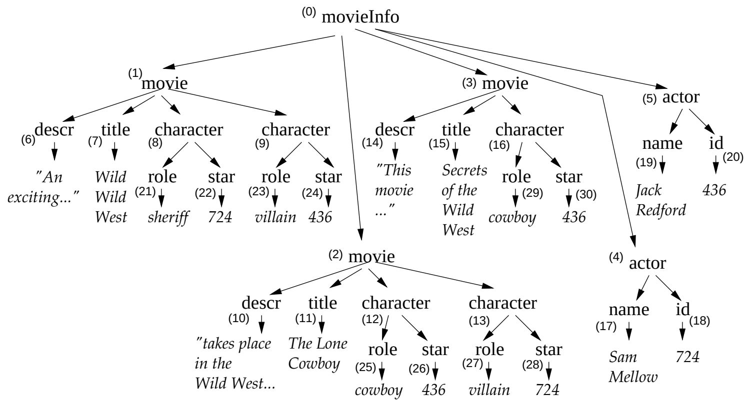

Fig. 1. An XML document describing information about movies.

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 useful. 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 to the user. Thus, it is not necessary to first query the database for its structure before searching for the desired information.

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 2). 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.

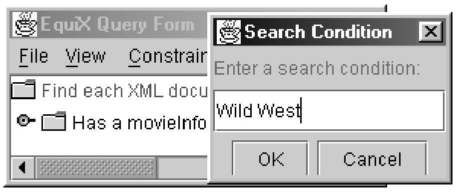

Fig. 2. Minimal query that finds documents containing the phrase "Wild West".

In Figure 3 an expanded concrete query is depicted. This query was formulated from the DTD presented in Section 2. It retrieves the title and description of Wild West movies in which Redford does not star as a villain. Intuitively, answering this query is a two part process. First, *search* for Wild West movies. The phrase "Wild West" may appear anywhere below the movie element. This is similar to a search in a search engine. Second, *query* the movies to find those in which Redford does not play as a villain. This condition is rather exact. It specifies 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 our abstract syntax. 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 $\mathcal{C}$.

Definition 2 (Abstract Query). *An abstract query is a rooted directed tree T augmented by four functions and an output set, denoted (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;*
- $o : N \to \{\wedge, \vee\}$ *is an* operator function *that associates each node with a logical operator;*

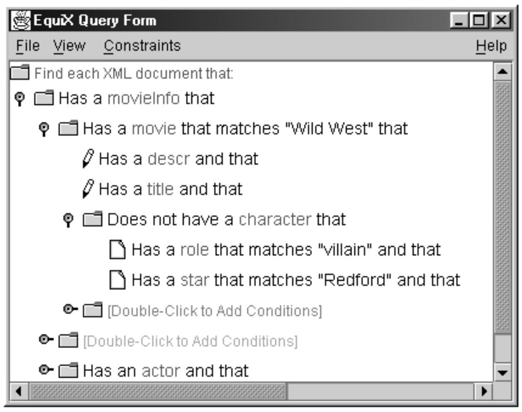

Fig. 3. Query for the titles and descriptions of movies in which Redford isn't a villain.

- $q : E \rightarrow \{\exists, \forall\}$ *is a* quantification function *that associates each edge with a quantifier;*
- $O \subseteq N$ *is a set of* projected nodes, *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 an 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. Translating one to the other is straightforward and is not presented due to space limitations.

The concrete query in Figure 3 can be represented by the abstract query in Figure 4. The string matching functions are specified in italics next to the nodes.

Black nodes are output nodes. In the sequel, unless otherwise specified, the term *query* will refer to an abstract query.

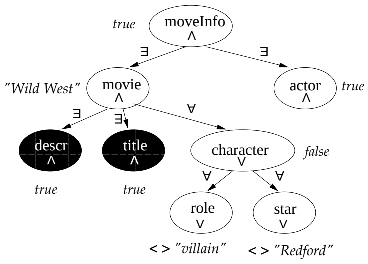

Fig. 4. Abstract query for the concrete query in Figure 3. Output nodes are black.

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 format of the result (Criterion 1). In EquiX, by defining the set O, the user only specifies what information should 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). 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 [3] we show how a DTD for the result documents can be computed. Thus, requerying of results is possible in EquiX (Criterion 3). In Section 5 we show that EquiX queries can be evaluated in polynomial time, and thus, EquiX meets Criterion 4.

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 similarly.

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 3 (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*

1. **Roots Match:** *$\mu(r_Q) = \{r_X\}$;*
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 $n_X \neq r_X$, 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 insures 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 is 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

- If n_X is an atomic node, then $t(n_X) = l_X(n_X)$;
- Otherwise, $t(n_X)$ is a 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 1. 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

- e is an existential-edge and there is a child n'_X of n_X such that n'_X matches n'_Q and $n'_X \in \mu(n'_Q)$.
- e is a universal-edge and 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 (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 mapping 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] 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. We will not elaborate on this point any further.

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 1. Recall the query in Figure 4 and the document in Figure 1. 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. One satisfying matching of the document to the query is: $\mu(\text{movieInfo}) = \{0\}$, $\mu(\text{movie}) = \{2\}$, $\mu(\text{descr}) = \{10\}$, $\mu(\text{title}) = \{11\}$, $\mu(\text{character}) = \{12, 13\}$, $\mu(\text{role}) = \{25, 27\}$, $\mu(\text{star}) = \{26, 28\}$, $\mu(\text{actor}) = \{4\}$.

We presented several 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, then $(\mu \cup \mu')(n_Q) := \mu(n_Q) \cup \mu'(n_Q)$. There may be an exponential number of matchings of a given document to a given query. Note, however, that the following holds.

Proposition 1 (Union of Matchings). *Let X be an XML document and let Q be a query. Let $\mathcal{M}$ be the set of all matchings of X to Q. Then the union of all the matchings in $\mathcal{M}$ is a matching. Formally, $(\bigcup_{\mu \in \mathcal{M}} \mu) \in \mathcal{M}$.*

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.

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 *descendents* of these nodes. The ancestors insure 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 descendents 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

- $N_{\text{out}} := \{n_X \in N_X \mid (\exists n_O \in O)(\exists \mu \in \mathcal{M})\ n_X \in \mu(n_O)\}$;
- $N_{\text{anc}} := \{n_X \in N_X \mid (\exists n'_X \in N_{\text{out}})\ n_X$ is an ancestor of $n'_X\}$;

– $N_{\text{desc}} := \{n_X \in N_X \mid (\exists n'_X \in N_{\text{out}})\ n_X$ is an descendent of $n'_X\}$.

Note that nodes in N_{out} are document nodes that correspond to projected nodes in Q. The ancestors (descendents) of the nodes in N_{out} are in N_{anc} (N_{desc}). We call N_R the *output set* of X with respect to Q.

The result of applying the query in Figure 4 to the document in Figure 1 is depicted in Figure 5. Note that the values of "descr" and "title" are grouped by "movie". This follows naturally from the structure of the original document.

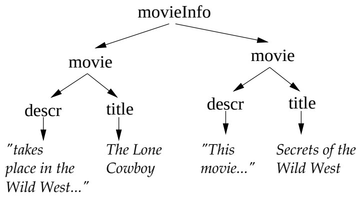

Fig. 5. Result of applying the query in Figure 4 to the document in Figure 1.

5 Query Evaluation

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). Evaluating Q on the database results in the set of documents created by evaluating Q on each document in S.

There may be an exponential number of matchings of a query to a document. Concrete queries contain both quantification and negation. 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 two variable first-order query. 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). Thus, EquiX has polynomial evaluation time (Criterion 4).

Consider a query Q with nodes N_Q and a document X with nodes N_X. Let $|D|$ be the size of the data in document X, i.e., the size of X when ignoring X's meta-data. Let $C(m)$ be an upper-bound on the runtime of computing a string-matching constraint on a string of size m. The theorem below gives the complexity of query evaluation. Its proof can be found in [3].

Theorem 1 (Polynomial Complexity). *The result of evaluating the query Q on the document X can be computed in time $O(|N_X| \cdot |N_Q| \cdot (|N_Q| \cdot |N_X| + C(|D|)))$.*

Query evaluation generates a set of documents. A query is formulated by exploring a DTD. Thus, in order to allow *iterative querying* or *requerying of results*, a DTD for the resulting documents must be defined. In [3] we present a polynomial procedure that computes a DTD for the resulting documents of a given query. This DTD is linear in the size of the DTD from which the query was originated. The compactness of the result DTD makes the requerying process simpler, since requerying entails exploring the result DTD. Thus, EquiX fulfills the search language requirement of ability to perform requerying (Criterion 3).

6 Conclusion

Several XML query languages have been proposed recently, such as XQL [7], XML-QL [4] and Lorel [5]. They are powerful in their querying ability. However, they do not fulfill some of our search language requirements. In these languages 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 2). Most importantly, none of these languages guarantee polynomial evaluation under combined complexity (Criterion 4).

EquiX fulfills all the requirements presented in Section 1. It also has a user-friendly concrete syntax. In our language, complex queries with negation, quantification and logical expressions can be expressed. We have also extended EquiX to allow aggregation and limited regular expressions [3]. As future work, we plan to extend the ability of querying ontologies [3] and to allow use of more complex regular expressions. We also plan to refine EquiX with the ability to deal with incomplete information and to add a metric for ranking the results that considers both the data and the meta-data.

References

1. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0, 1998. Available at `http://www.w3.org/XML`.
2. S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX—Easy querying in XML databases. In *Proc. 2nd International Workshop on the Web and Databases*, Philadelphia, (Pennsylvania, USA), June 1999.
3. S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. Equix—A search and query language for XML. Technical Report 2000-28, Departement Computer Science, Hebrew University, Jerusalem, Israel, 2000. Available at `http://www.cs.huji.ac.il/leibniz/research/2000.html`.
4. 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/NOTE-xml-ql`.
5. R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proc. 2nd International Workshop on the Web and Databases*, Philadelphia, (Pennsylvania, USA), June 1999.
6. B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. A framework for navigation-driven lazy mediators. In *Proc. 2nd International Workshop on the Web and Databases*, Philadelphia, (Pennsylvania, USA), June 1999.
7. J. Robie, J. Lapp, and D. Schach. XML query language (XQL), 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.