Answering XPath Queries over Networks by Sending Minimal Views

Keishi Tajima Yoshiki Fukui

Japan Advanced Institute of Science and Technology (JAIST)
Asahidai, Tatsunokuchi, Ishikawa 923-1292 Japan, {tajima, y-fukui}@jaist.ac.jp

Abstract

When a client submits a set of XPath queries to a XML database on a network, the set of answer sets sent back by the database may include redundancy in two ways: some elements may appear in more than one answer set, and some elements in some answer sets may be subelements of other elements in other (or the same) answer sets. Even when a client submits a single query, the answer can be self-redundant because some elements may be subelements of other elements in that answer. Therefore, sending those answers as they are is not optimal with respect to communication costs. In this paper, we propose a method of minimizing communication costs in XPath processing over networks. Given a single or a set of queries, we compute a minimal-size view set that can answer all the original queries. The database sends this view set to the client, and the client produces answers from it. We show algorithms for computing such a minimal view set for given queries. This view set is optimal; it only includes elements that appear in some of the final answers, and each element appears only once.

1 Introduction

Recently, XML has become a standard data format for information exchange and dissemination over the Internet. There have been many researches on various styles of XML information services on networks, such as on-line XML databases that provide interactive querying interfaces over the Internet, continuous query systems [21, 9, 2], and XML streaming systems [22, 5, 17, 25]. A continuous query system is a push-based information service, in which the users first register their queries to the system. Then, the system monitors the changes in its data, and when data matching the user queries become available, it is delivered to the corresponding users. In XML streaming systems, a server transmits a XML stream to the clients, and the clients monitor the stream to detect the data of interest to them.

Most of those systems use some kind of query language. Some of them use their own languages, but recently a language called XPath [12, 13] has become very popular. Although it was originally designed as a component of other standards, it is now also used as a stand-alone query language for many XML information systems because of its simplicity and yet enough expressive power [2, 8, 17, 25]. XML data is essentially a tree with node labels, and XPath is a tree pattern language, which extracts from a XML data a set of subtrees rooted by nodes that match the tree pattern. XPath can only extract a whole subtree rooted by some element; it never adds or trims edges, nor modifies labels.

XML information services can be classified into two categories: those that process queries on the server side, such as on-line XML databases and continuous query systems, and those that process queries on the client side, such as XML streaming systems. In the former, only necessary information is sent over networks from the server to the clients, and therefore, they are more efficient with respect to communication costs.

Even in the server-side approach, however, the communication cost is not always optimal. For example, if a client submits two queries to a database on a network, and the two answer sets to them have some data in common, sending those two answer sets separately to the client is not optimal in the sense that some data are sent twice. For example, suppose a client submits two queries asking:

- abstracts of papers including “XML” in their titles
- entire papers including both “XML” and “XPath” in their titles

to an on-line digital library or a continuous query system. If some paper includes “XML” and “XPath” in its title, its abstract appears in the answer sets twice, once as an answer to the first query, and once as a subelement of an answer to the second query. Hence, sending the two answer sets to

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

the client over the network is not optimal with respect to the communication cost.

Even when a client submits a single XPath query, the answer can be self-redundant, i.e., some elements in the answer set may be subelements of other elements in that answer set. For example, suppose a client issues the query:

- retrieve chapters, sections, or subsections that have the word “XML” in their headings.

Then, if a book has a section with the heading “XML” and its subsection with the heading “XML queries”, that subsection is sent to the client twice, once as an answer and once as a subelement of another answer. Answers to XPath queries are self-redundant very often. Notice that the same situation also occurs if a user issues a single query asking the union of the two queries in the previous example.

In the worst case, if a client submits a query “retrieve any subtree of the database tree,” the result sent to the client can be far larger than the database itself. Self-redundancy of answers comes from the characteristic of XPath queries in which an answer to a query is a set of subtrees of the database tree, and a member of an answer set may be a subtree of another member of the same answer set. (However, a similar phenomenon can occur in other databases as well because it is quite usual that a query language can create an answer that is bigger than the database itself. For example, in relational databases, one can query the product of all the relations in the database [11].)

If the server and the client agree on some encoding or protocol, we can avoid such redundancy in query answers in various ways, e.g., embedding “pointers” in the answers. In this research, however, we assume the server is a service on the Internet provided by someone else, and all we can do is to submit XPath queries and get answers.

Even in such an environment, it is possible to minimize the size of the data sent over the network in the examples above. In the first example, we can minimize it by submitting the following two queries instead of the original ones:

- retrieve abstracts of papers including the word “XML” in their titles, but not “XPath”, and

- retrieve entire papers including the words “XML” and “XPath” in their titles.

The server sends the answers to those queries, and then, the client can produce the answers to the original queries from those two results. In this scenario, the data sent over the network is optimal because it only includes data that appears in either (or both) of the final answers without duplication. In the same way, in the second example, the client should submit the following query:

- retrieve chapters, sections, or subsections that have “XML” in their headings, but have no ancestor with “XML” in its heading.

Then, the client can extract all the answers to the original query from the answers to this query. The data sent over the network in this example is again optimal.

In this way, we can sometimes optimize the communication cost by leaving a part of the query evaluation to the client, rather than fully evaluating queries at the server. By generalizing these examples, we study the following problem in this paper: *Given a single or a set of XPath queries, we compute another set of queries such that:*

- *we can produce the answers to the original queries from their answers, and*

- *the total size of their answers is minimal.*

In other words, we compute a minimal view set that can answer all the original queries. In this paper, we show algorithms for computing such a view set. Notice that a view set that includes all the information in the final answers does not necessarily guarantee we can correctly extract them. This is because some context in the database may be lost in the views. We will show some examples later.

If the server supports a full-fledged query language like XQuery, we can write queries that embed markers in views so that the client can easily extract the answers to the original queries. XPath, however, can only extract a set of subtrees of the database tree without modification, thus making the problem non-trivial; nevertheless, it is also this property that makes XPath efficiently processable, and it is the reason why many researches on large-scale information services adopt XPath [2, 8, 17, 25].

The techniques shown in this paper can be used in several ways. One way is to embed them in an intelligent querying agent, which resides at the client site. The agent transforms the user queries before submitting them, and extracts the answers from the views received from the server. Another approach is to embed them in a proxy server which resides between a continuous query server on the Internet and its users on the local network. Those users register queries to the proxy server, and the proxy server registers transformed queries to the server. By this, if many users register queries with overlapping answers, we can optimize the communication cost over the Internet.

In the next section, we explain the fragment of XPath we use in this paper. Next, we show some examples to clarify the problem and its inherent difficulties, and then, we formulate the problem. In the following three sections, we show our algorithm in three steps. We begin with an algorithm for non-recursive queries. (The meaning of non-recursive/recursive queries is explained later.) Second, we show an algorithm for a single recursive query, and finally we show an algorithm for the general case. Then, we discuss related work. The final section summarizes the paper and briefly discusses the practicality of our method.

2 XPath

As mentioned above, XPath is evaluated on a XML tree, and returns a set of subtrees rooted by nodes matching the pattern. Here, we assume that a query answer is given in the form of a XML tree rooted by a node labeled Ans that has all the matching subtrees as its children (as in some

XPath processors, e.g., Xalan [27]). For example, when a query answer is the following set of three subtrees:

$$\{\langle \mathsf{a}\rangle \dots \langle /\mathsf{a}\rangle, \quad \langle \mathsf{b}\rangle \dots \langle /\mathsf{b}\rangle, \quad \langle \mathsf{b}\rangle \dots \langle /\mathsf{b}\rangle\}$$

it is given as an XML tree in a form:

$$\langle \mathsf{Ans}\rangle \quad \langle \mathsf{a}\rangle \dots \langle /\mathsf{a}\rangle \quad \langle \mathsf{b}\rangle \dots \langle /\mathsf{b}\rangle \quad \langle \mathsf{b}\rangle \dots \langle /\mathsf{b}\rangle \quad \langle /\mathsf{Ans}\rangle.$$

In this paper, we use a fragment of XPath language that only includes its main features. The syntax of the language is defined as follows:

$$\begin{array}{lll} q & ::= & /p \mid //p \mid q \cup q \mid q - q \\ p & ::= & a \mid \overline{\{a_1, \dots, a_n\}} \mid * \mid p/p \mid p//p \mid p[p] \mid p\overline{[p]} \end{array}$$

A query q is either an absolute location path (in XPath terminology) of the form $/p$ or $//p$, the union of two queries $q \cup q$, or the difference of two queries $q - q$. An absolute location path $/p$ matches nodes which are reachable from the root through paths matching a relative location path (in XPath terminology) p. On the other hand, $//p$ matches nodes which are reachable through paths matching p starting from any nodes. $q \cup q$ and $q - q$ are the ordinary set union and the ordinary set difference.

A relative location path p is composed of the following constructs. a is a label test that matches nodes with a label a, and a negative label test $\overline{\{a_1, \dots, a_n\}}$ matches nodes with a label other than $a_1, \dots, a_n$. $*$ is a wild card that matches nodes with any labels. p/p is a concatenation of two location paths. For example, /a/$*$ matches nodes with any label which are children of the "a" node at the root of the database tree. $p_1//p_2$ is also a concatenation, but it does not require a path matching p_2 appears immediately beneath a path matching p_1. For example, /a/$*$//b matches any "b" nodes which are descendants of the nodes matching the previous query /a/$*$. // represents a restricted form of recursion, and we call queries with // (without //) recursive queries (non-recursive queries, respectively).

$p_1[p_2]$ is called a predicate expression, and it matches nodes which are reachable through paths matching p_1, and also have at least one path matching p_2 beneath them. For example, //a[b/c] matches "a" nodes at any depth that has a child node "b" which, in turn, has a child node "c". Similarly, //a[b][c] matches "a" nodes at any depth that have both "b" children and "c" children. $p_1\overline{[p_2]}$ is a negative predicate and it matches nodes that are reachable through paths matching p_1 but have no path matching p_2 beneath them.

The definition above does not include the intersection operation $q_1 \cap q_2$, but it can be computed by $q_1 - (q_1 - q_2)$. Complementation of q can also be computed by $//* - q$. If we assume a finite set of labels, $\overline{\{a_1, \dots, a_n\}}$ and $*$ add no expressive power to the language, but here we assume an infinite set of labels. Notice that $\overline{\{a_1, \dots, a_n\}}$ has more power than the combination of $\overline{\{a\}}$ and $\cap$. For example, $//\overline{\{\mathsf{a}, \mathsf{b}\}}//\mathsf{c}$ is not equivalent to $(//\overline{\{\mathsf{a}\}}//\mathsf{c}) \cap (//\overline{\{\mathsf{b}\}}//\mathsf{c})$.

$q - q$ is supported in XPath 2.0 [13]. In XPath 1.0 [12], it is not directly supported, but we can express it by using a negative eq-join with identity-equality. Negative eq-join can be expressed by absolute location paths in negative predicates, and identity-equality can be expressed by using built-in count function as shown in [24]. Similarly, $\overline{\{a_1, \dots, a_n\}}$ is not directly supported in XPath standard, but we can express it by $*$[not(self::a_1)]...[not(self::a_n)]. Negative predicates are expressed by p[not(p)]. For more details of the XPath standards, please refer to [12, 13].

3 Problem Analysis

In this section, we show some motivating examples in order to clarify what is the problem in XPath processing over a network, and what are the difficulties in it.

3.1 Examples with Non-recursive Queries

First, we consider examples that only include non-recursive queries. Below are two simple examples of a set of XPath queries that cause redundancy in their answers:

$$\left\{ \begin{array}{ll} Q_1: & /\mathsf{a}/* \\ Q_2: & /\mathsf{a}/\mathsf{b} \end{array} \right. \qquad \left\{ \begin{array}{ll} Q_3: & /\mathsf{a}/\mathsf{b}[\mathsf{c}] \\ Q_4: & /\mathsf{a}/\mathsf{b}[\mathsf{d}] \end{array} \right.$$

In the example on the left side, the answer to Q_2 is a subset of Q_1, and therefore, sending the answers to Q_1 and Q_2 separately is not optimal with respect to communication costs. In this case, a simple solution is that we submit only Q_1 to the server, and produce the answer to Q_2 at the client side by extracting only b elements from the answer to Q_1. Because we assume that the answer to a query is given in a form of a XML tree rooted by Ans node whose children are answer elements, we can do that by evaluating a query /Ans/b against the answer to Q_1. In the rest of the paper, we write this in the following syntax:

$$Q_2 \leftarrow (Q_1, /\mathsf{Ans}/\mathsf{b})$$

On the other hand, in the case of Q_3 and Q_4, their answers overlap only partially. In this case, we can submit the query below instead of Q_3 and Q_4:

$$\left\{ \; Q_{3\cup 4}: \quad /\mathsf{a}/\mathsf{b}[\mathsf{c}] \cup /\mathsf{a}/\mathsf{b}[\mathsf{d}] \right.$$

and we can extract the answer to Q_3 and Q_4 at the client in the following way:

$$\begin{array}{lll} Q_3 & \leftarrow & (Q_{3\cup 4}, /\mathsf{Ans}/\mathsf{b}[\mathsf{c}]) \\ Q_4 & \leftarrow & (Q_{3\cup 4}, /\mathsf{Ans}/\mathsf{b}[\mathsf{d}]) \end{array}$$

Similar situations are caused by union, difference, and negative label tests, such as:

$$\left\{ \begin{array}{ll} Q_5: & /\mathsf{a}/\mathsf{b} \cup /\mathsf{a}/\mathsf{c} \\ Q_6: & /\mathsf{a}/\mathsf{b} \end{array} \right. \qquad \left\{ \begin{array}{ll} Q_7: & /\mathsf{a}/\overline{\{\mathsf{b}\}} \\ Q_8: & /\mathsf{a}/\overline{\{\mathsf{c}\}} \end{array} \right.$$

Those two cases can be handled in the same way as the two examples above, respectively.

In some cases, however, we cannot extract the answer to some query from the answer to another query even if the former is a subset of the latter. For example, suppose we have two queries below:

$$\left\{ \begin{array}{ll} Q_9: & /\mathsf{a}/*/\mathsf{c} \\ Q_{10}: & /\mathsf{a}/\mathsf{b}/\mathsf{c} \end{array} \right.$$

The answer to Q_9 is a superset of the answer to Q_{10}. In this case, however, only given the answer to Q_9 of the form:

$$\langle\text{Ans}\rangle \quad \langle\text{c}\rangle\ldots\langle\text{/c}\rangle \quad \ldots \quad \langle\text{c}\rangle\ldots\langle\text{/c}\rangle \quad \langle\text{/Ans}\rangle$$

we cannot tell which c elements in this answer are to be included in the answer to Q_{10} because we cannot know the labels of their parents in the original database tree. In this way, some context information in the database may be lost in query answers. In this case, we can minimize the communication cost, i.e., the total size of the data sent over the network, by submitting the following two queries:

$$\begin{cases} Q_{9-10}: & \text{/a/}\overline{\{\text{b}\}}\text{/c} \\ Q_{10}: & \text{/a/b/c} \end{cases}$$

The answer to Q_9 can be produced at the client side by taking union of the answers to Q_{9-10} and Q_{10} as follows:

$$\begin{array}{lcl} Q_9 & \leftarrow & (Q_{9-10}, \text{/Ans/}*) \\ Q_9 & \leftarrow & (Q_{10}, \text{/Ans/}*) \end{array}$$

The pair of Q_{9-10} and Q_{10} is optimal with respect to the communication cost because their answers only include data that appear in the final answers without duplication.

Similarly, if given two intersecting queries below:

$$\begin{cases} Q_{11}: & \text{/a/}\overline{\{\text{b}\}}\text{/d} \\ Q_{12}: & \text{/a/}\overline{\{\text{c}\}}\text{/d} \end{cases}$$

then, we should submit the following queries:

$$\begin{cases} Q_{11-12}: & \text{/a/c/d} \\ Q_{11\cap 12}: & \text{/a/}\overline{\{\text{b,c}\}}\text{/d} \\ Q_{12-11}: & \text{/a/b/d} \end{cases}$$

and produce the final answers in the following way:

$$\begin{array}{lcl} Q_{11} & \leftarrow & (Q_{11-12}, \text{/Ans/}*) \\ Q_{11} & \leftarrow & (Q_{11\cap 12}, \text{/Ans/}*) \\ Q_{12} & \leftarrow & (Q_{11\cap 12}, \text{/Ans/}*) \\ Q_{12} & \leftarrow & (Q_{12-11}, \text{/Ans/}*) \end{array}$$

This is more efficient with respect to the communication cost than submitting Q_{11} and Q_{12}, which results in sending elements in their intersection twice.

In the example above, the source of the redundancy are elements matching more than one query. Redundancy also arises when some answers also appear as subelements of other answers. Shown below are two simple examples:

$$\begin{cases} Q_{13}: & \text{/a/}* \\ Q_{14}: & \text{/a/b/c} \end{cases} \qquad \{\ Q_{15}: \ \text{/a/}* \cup \text{/a/b/c}$$

In the case of Q_{15}, its answer can be self-redundant, i.e., some elements in its answer set may also appear as subelements of other elements in the answer set. In these cases, we should submit the following queries, respectively:

$$\{\ Q_{13}: \ \text{/a/}* \qquad \{\ Q_{15}^{\top}: \ \text{/a/}*$$

and extract the final answers in the following way:

$$Q_{14} \leftarrow (Q_{13}, \text{/Ans/b/c}) \qquad \begin{array}{l} Q_{15} \leftarrow (Q_{15}^{\top}, \text{/Ans/}*) \\ Q_{15} \leftarrow (Q_{15}^{\top}, \text{/Ans/b/c}) \end{array}$$

In the examples above, we can extract elements matching /a/b/c from the answer to /a/$*$ because the answer to /a/$*$ includes enough context information for /a/b/c. In general, however, some context information may be lost in query answers as explained before, and we may need to submit up to three queries, as shown in the example below:

$$\begin{cases} Q_{16}: & \text{/a/}*\text{/c} \\ Q_{17}: & \text{/a/b/c/d} \end{cases} \qquad \begin{cases} Q_{18}: & \text{/a/}\overline{\{\text{b}\}}\text{/d} \\ Q_{19}: & \text{/a/}\overline{\{\text{c}\}}\text{/d/e} \end{cases}$$

Here, we should submit the following set of queries:

$$\begin{cases} Q_{16-17}: & \text{/a/}\overline{\{\text{b}\}}\text{/c} \\ Q_{16\cap 17}: & \text{/a/b/c} \end{cases} \qquad \begin{cases} Q_{18-19}: & \text{/a/c/d} \\ Q_{18\cap 19}: & \text{/a/}\overline{\{\text{b,c}\}}\text{/d} \\ Q_{19-18}: & \text{/a/b/d/e} \end{cases}$$

and produce the final answers in the following way:

$$\begin{array}{l} Q_{16} \leftarrow (Q_{16-17}, \text{/Ans/}*) \\ Q_{16} \leftarrow (Q_{16\cap 17}, \text{/Ans/}*) \\ Q_{17} \leftarrow (Q_{16\cap 17}, \text{/Ans/}*\text{/d}) \end{array} \qquad \begin{array}{l} Q_{18} \leftarrow (Q_{18-19}, \text{/Ans/}*) \\ Q_{18} \leftarrow (Q_{18\cap 19}, \text{/Ans/}*) \\ Q_{19} \leftarrow (Q_{18\cap 19}, \text{/Ans/}*\text{/e}) \\ Q_{19} \leftarrow (Q_{19-18}, \text{/Ans/}*) \end{array}$$

If we want to make the number of submitted queries as small as possible, in the example of Q_{18} and Q_{19}, we can merge Q_{18-19} with Q_{19-18} into a query (/a/c/d)$\cup$(/a/b/d/e) because we can extract the answers to Q_{18-19} and Q_{19-18} from its answer by /Ans/e and /Ans/d. It is also possible to merge $Q_{18\cap 19}$ and Q_{19-18} instead. Although it is not difficult to detect such cases, in this paper, because of space limitations, we only consider the elimination of data redundancy, and do not discuss the minimization of the number of queries. Therefore, even in the previous example of Q_{13} and Q_{14}, where we need to submit only Q_{13}, the algorithm we show later produces two queries.

3.2 Examples with Recursive Queries

The examples shown so far included only non-recursive queries, i.e., queries without //. When queries include // or union operator $\cup$, the redundancy in the answers occurs even when a user submits a single query. We have already shown an example with $\cup$. Below is an example with //:

$$\{\ Q_{20}: \text{//a}$$

This query retrieves all the subtrees rooted by “a” nodes in the database tree. Therefore, if some “a” nodes occur as descendants of other “a” nodes, the subtrees rooted by those descendant “a” are sent more than once over the network. In this way, answer sets to recursive XPath queries are self-redundant by nature because of the nested structure of XML.

In this case, we can minimize the size of the data sent over the network by submitting the query below to the server:

$$\{\ Q_{20}^{\top}: \text{//a} - \text{//a//}*$$

This query retrieves “a” nodes that occur as the first “a” node in each path from the root. Then, we can produce the answer to the original query in the following way:

$$Q_{20} \quad \leftarrow \quad (Q_{20}^{\top}, \text{/Ans//a})$$

Extraction of answers that are descendant of other answers can be more complicated. Suppose we have a query:

$$\{\ Q_{21} : \quad \text{//a/b/a/b}$$

Then, we should submit the query below:

$$\{\ Q_{21}^{\top} : \quad \text{//a/b/a/b} - \text{//a/b/a/b//}*$$

$-$//a/b/a/b//$*$ at the tail eliminates the self-redundancy in the answer. In order to extract all the answers to Q_{21} from the answer to $Q_{21}^{\top}$, we need three queries shown below:

$$\begin{array}{lcl} Q_{21} & \leftarrow & (Q_{21}^{\top}, \text{/Ans/b}) \\ Q_{21} & \leftarrow & (Q_{21}^{\top}, \text{/Ans/b/a/b}) \\ Q_{21} & \leftarrow & (Q_{21}^{\top}, \text{/Ans//a/b/a/b}) \end{array}$$

Because a b element in the answer to $Q_{21}^{\top}$ is an element that has matched //a/b/a/b, if it has a path a/b beneath it, that grandchild b node was also matched //a/b/a/b in the database. Therefore, we need /Ans/b/a/b shown above. The computation of how to extract descendant answers is similar to the computation of the prefix function in the classic Knuth-Morris-Pratt algorithm for substring search [19].

For recursive queries, we sometimes need more than one query even when a client submits a single union-free query. For example, suppose we have the query below:

$$\{\ Q_{22} : \quad \text{//a/}*\text{/}*$$

which retrieves grandchildren of “a” nodes occurring at any level in the database tree. In this case, the single query and two procedure below are not sufficient:

$$\{Q_{22}^{\top} : \text{//a/}*\text{/}* - \text{//a/}*\text{/}*\text{//}* \qquad \begin{array}{l} Q_{22} \leftarrow (Q_{22}^{\top}, \text{/Ans/}*) \\ Q_{22} \leftarrow (Q_{22}^{\top}, \text{/Ans//a/}*\text{/}*) \end{array}$$

because these two procedures cannot extract answers that are children of some answers to $Q_{22}^{\top}$. For example, if there is a path /a/a/b/c in the database tree, both the node b and the node c match Q_{22}, and only b is included in the answer to $Q_{22}^{\top}$. Only from the path /Ans/b/c in the answer to $Q_{22}^{\top}$, however, we cannot know the labels of the parent of b, and therefore, we cannot tell if we should extract the c node.

In this case, we can correctly extract the answer to Q_{22} while minimizing the communication cost by submitting the following two queries:

$$\left\{ \begin{array}{ll} Q_{\mathsf{a}}^{\top} : & \text{//a/a/}* - \text{//a/}*\text{/}*\text{//}* \\ Q_{\overline{\{\mathsf{a}\}}}^{\top} : & \text{//a/}\overline{\{\mathsf{a}\}}\text{/}* - \text{//a/}*\text{/}*\text{//}* \end{array} \right.$$

and by extracting the answer to Q_{22} in the following way:

$$\begin{array}{lcl} Q_{22} & \leftarrow & (Q_{\mathsf{a}}^{\top}, \text{/Ans/}*) \\ Q_{22} & \leftarrow & (Q_{\mathsf{a}}^{\top}, \text{/Ans/}*\text{/}*) \\ Q_{22} & \leftarrow & (Q_{\mathsf{a}}^{\top}, \text{/Ans//a/}*\text{/}*) \\ Q_{22} & \leftarrow & (Q_{\overline{\{\mathsf{a}\}}}^{\top}, \text{/Ans/}*) \\ Q_{22} & \leftarrow & (Q_{\overline{\{\mathsf{a}\}}}^{\top}, \text{/Ans//a/}*\text{/}*) \end{array}$$

4 Problem Formulation

Now we formulate the problem we study in this paper. First, when an XML element e_1 is a descendant of another XML element e_2, we write $e_1 \prec e_2$. We also write $e_1 \preceq e_2$ to mean $e_1 \prec e_2$ or $e_1 = e_2$. Next, for a bag B of elements (we use a bag because data sent over a network may include the same element more than once), we define $\mathcal{E}(B)$, the subelement-enumeration of B, as follows:

$$\mathcal{E}(B) \equiv \bigcup_{e \in B}^{b} \{s \mid s \preceq e\}$$

where $\bigcup_{e \in B}^{b}$ is the iteration of the bag union operation $\cup^b$ for all the elements in B including multi-occurrences. Using $\mathcal{E}(B)$, we define a partial order $\subseteq_{\mathcal{E}}$ as follows:

$$B_1 \subseteq_{\mathcal{E}} B_2 \equiv \mathcal{E}(B_1) \subseteq^b \mathcal{E}(B_2)$$

where $\subseteq^b$ is the bag inclusion. Next, $Q(t)$ denotes the result of the evaluation of a query Q against an XML tree t. Then, we say a set of queries $\{V_1, \ldots, V_m\}$ is a view set that can answer a query Q iff:

$$(\exists q_1, \ldots, q_m)(\forall t) Q(t) = q_1(V_1(t)) \cup \ldots \cup q_m(V_m(t))$$

Now, we formulate the problem as follows:
Minimal View Selection Problem: Given a set of XPath queries $\{Q_1, \ldots, Q_n\}$, we compute a view set (i.e., another set of XPath queries) $\mathcal{V} = \{V_1, \ldots, V_m\}$, such that:

1. $\mathcal{V}$ can answer all of $Q_1, \ldots, Q_n$, and
2. among those satisfying 1, $V_1(t) \cup^b \ldots \cup^b V_m(t)$ is minimal under $\subseteq_{\mathcal{E}}$ for any t. □

In the following three sections, we show algorithms to compute such minimal view sets. We begin with an algorithm for an arbitrary number of non-recursive queries, then show an algorithm for a single recursive query, and finally explain an algorithm for the general case, i.e., for an arbitrary number of recursive queries. In this paper, for the sake of brevity, we restrict the input of our algorithms to a language without union ($q \cup q$) and difference ($q - q$). We can, however, extend our algorithms for the language including them. Although we forbid those operations in the input, we use them in the output when we have recursive queries as shown in the later sections.

5 Algorithm for Non-Recursive Queries

This section explains an algorithm that computes a minimal view set for a given set of non-recursive XPath queries.

5.1 Intuition behind the Algorithm

First, we explain the intuition behind our algorithm, and show a simpler algorithm only for two queries.

Suppose we are given a set of non-recursive queries $Q_1, \ldots, Q_n$. In the simplest case, if all of them have the same length, i.e., the same number of /, then an element in the answer to some Q_i cannot be a subelement of elements in the answers to the other queries. In that case, the problem is rather easy. The following set of queries is a minimal view that can answer to $Q_1, \ldots, Q_n$:

$$\{V(S) \mid S \neq \emptyset,\ S \subseteq \{1, \ldots, n\}\}$$

where

$$V(S) = \bigcap_{i \in S} Q_i - \bigcup_{i \in \{1,\ldots,n\}-S} Q_i$$

From this view set, we can extract answers by executing:

$$Q_j \leftarrow (V(S),\ \text{/Ans/}*)$$

for every j, S s.t. $j \in S$. For example, if $n = 2$, the minimal view is:

$$\{Q_1 - Q_2,\ Q_1 \cap Q_2,\ Q_2 - Q_1\}$$

and we can extract the answers to Q_1 and Q_2 by

$$\begin{array}{lcl} Q_1 & \leftarrow & (Q_1 - Q_2, \text{/Ans/}*) \\ Q_1 & \leftarrow & (Q_1 \cap Q_2, \text{/Ans/}*) \\ Q_2 & \leftarrow & (Q_1 \cap Q_2, \text{/Ans/}*) \\ Q_2 & \leftarrow & (Q_2 - Q_1, \text{/Ans/}*) \end{array}$$

In general, however, the length of $Q_1, \ldots, Q_n$ are not the same, and some element in one answer may be a subelement of elements in other answers. To deal with it, we define four set operations, generalized upper intersection $\cap_\succ$, generalized lower intersection $\cap_\prec$, generalized difference $-_\preceq$, and self-redundancy elimination $^\top$, as follows:

$$\begin{array}{r} S_1 \cap_\succ S_2 \equiv \{e_1 \in S_1 \mid (\exists e_2 \in S_2)\ e_1 \succ e_2\} \\ S_1 \cap_\prec S_2 \equiv \{e_1 \in S_1 \mid (\exists e_2 \in S_2)\ e_1 \prec e_2\} \\ S_1 -_\preceq S_2 \equiv \{e_1 \in S_1 \mid (\forall e_2 \in S_2)\ e_1 \not\preceq e_2 \wedge e_2 \not\preceq e_1\} \\ S^\top \equiv \{e \mid e \text{ is maximal in } S \text{ under } \preceq\} \end{array}$$

If we substitute $=$ for $\prec$, $\succ$ in those definitions, $\cap_\succ$, $\cap_\prec$, $-_\preceq$ fall back into the ordinary intersection and difference. Notice that $\cap_\succ$, $\cap_\prec$ are not symmetric. (They are actually instances of "filter" operator in [1], while $-_\preceq$ is an instance of generalized difference in [1].) If $S^\top \neq S$, we say S is self-redundant. Then, the following properties hold.

Proposition 1 *The following equation holds:*
$S_1 = (S_1 -_\preceq S_2) \cup (S_1 \cap_\succ S_2) \cup (S_1 \cap S_2) \cup (S_1 \cap_\prec S_2)$

Proof: l.h.s.$\supseteq$r.h.s. is obvious from the definition of $-_\preceq$, $\cap_\succ$, $\cap$, $\cap_\prec$. l.h.s.$\subseteq$r.h.s. is also obvious because every $e_1 \in S_1$ appears at least one of the four sets in r.h.s. □

Proposition 2 *If S_1, S_2 are not self-redundant, seven sets: $S_1 -_\preceq S_2$, $S_1 \cap_\succ S_2$, $S_1 \cap_\prec S_2$, $S_1 \cap S_2$, $S_2 \cap_\prec S_1$, $S_2 \cap_\succ S_1$, $S_2 -_\preceq S_1$ are disjoint with one another.*

Proposition 3 *If S_1, S_2 are not self-redundant, the seven sets above are also pairwise disjoint in the sense of $\cap_\succ$ and $\cap_\prec$ except for the four cases below:*

$$\begin{array}{rcl} (S_1 \cap_\succ S_2) \cap_\succ (S_2 \cap_\prec S_1) & = & (S_1 \cap_\succ S_2) \\ (S_1 \cap_\prec S_2) \cap_\prec (S_2 \cap_\succ S_1) & = & (S_1 \cap_\prec S_2) \\ (S_2 \cap_\succ S_1) \cap_\succ (S_1 \cap_\prec S_2) & = & (S_2 \cap_\succ S_1) \\ (S_2 \cap_\prec S_1) \cap_\prec (S_1 \cap_\succ S_2) & = & (S_2 \cap_\prec S_1) \end{array}$$

Proof Outline of Proposition 2, 3: It is easy to show that if any two of the seven sets have other intersections, it implies that either S_1 or S_2 is self-redundant. □

We also define those operations for queries, such as:

$$\begin{array}{rcl} Q_1 \cap_\succ Q_2 & \equiv & \{Q \mid (\forall t) Q(t) = Q_1(t) \cap_\succ Q_2(t)\} \\ Q_1 \cap_\prec Q_2 & \equiv & \{Q \mid (\forall t) Q(t) = Q_1(t) \cap_\prec Q_2(t)\} \\ Q_1 -_\preceq Q_2 & \equiv & \{Q \mid (\forall t) Q(t) = Q_1(t) -_\preceq Q_2(t)\} \\ Q_1^\top & \equiv & \{Q \mid (\forall t) Q(t) = Q_1(t)^\top\} \end{array}$$

They are defined by the set of equivalent queries, but we also use those notations to denote some representative queries for those equivalent classes in the rest of the paper. We also say Q is self-redundant if $(\exists t) Q(t) \neq Q^\top(t)$. Then, the following proposition holds for $Q^\top$.

Proposition 4 *Non-recursive union-free queries are never self-redundant.*

Proof: All the elements in the answer to a non-recursive union-free query appear at the same level in the database tree corresponding to the length of the query. Therefore, no element can be a subelement of the other. □

Therefore, Proposition 2 and 3 apply to the answers to non-recursive union-free queries. Then, the following property holds:

Proposition 5 *When given two non-recursive union-free queries Q_1 and Q_2, we can retrieve all the necessary elements for their answers without duplication by five queries $Q_1 -_\preceq Q_2$, $Q_1 \cap_\succ Q_2$, $Q_1 \cap Q_2$, $Q_2 \cap_\succ Q_1$, and $Q_2 -_\preceq Q_1$.*

Proof: Let S_1 and S_2 be the answers to Q_1 and Q_2. By Proposition 1, those five sets include all the elements in $S_1 \cup S_2$ except for those in $S_1 \cap_\prec S_2$ and $S_2 \cap_\prec S_1$. Elements in $S_1 \cap_\prec S_2$ and $S_2 \cap_\prec S_1$ are, however, included in $S_2 \cap_\succ S_1$ and $S_1 \cap_\succ S_2$ as subelements because of Proposition 3. In addition, by Proposition 2 and 3, those five sets include no duplication even in the sense of $\cap_\prec$ or $\cap_\succ$. □

This property suggests that the set of those five sets may work as a minimal view that can answer Q_1 and Q_2 (although this proposition just guarantees that all the necessary elements are included in those five sets, and does not guarantee that $S_1 \cap_\prec S_2$ and $S_2 \cap_\prec S_1$ can correctly be extracted from $S_2 \cap_\succ S_1$ and $S_1 \cap_\succ S_2$).

Following this observation, we can develop a simple algorithm for computing a minimal view set for two non-recursive union-free queries. Let Q_1 be $/p_1^1/\ldots/p_1^n$ and Q_2 be $/p_2^1/\ldots/p_2^m$ where p_i^j are expressions that do not include / nor //. We can assume $n \leq m$ w.l.o.g. We can compute the minimal view for Q_1 and Q_2 in the following way.

If $n = m$, $Q_1 \cap_\succ Q_2$, $Q_1 \cap_\prec Q_2$, $Q_2 \cap_\succ Q_1$, and $Q_2 \cap_\prec Q_1$ are empty queries, and the following three queries constitute a minimal view:

$$\begin{array}{rl} Q_1 -_\preceq Q_2 : & /p_1^1/\ldots/p_1^n - /p_2^1/\ldots/p_2^m \\ Q_1 \cap Q_2 : & /p_1^1/\ldots/p_1^n \cap /p_2^1/\ldots/p_2^m \\ Q_2 -_\preceq Q_1 : & /p_2^1/\ldots/p_2^m - /p_1^1/\ldots/p_1^n \end{array}$$

and we can extract the final answers as follows:

$$\begin{array}{lcl} Q_1 & \leftarrow & (Q_1 -_{\preceq} Q_2, \text{/Ans/}*) \\ Q_1 & \leftarrow & (Q_1 \cap Q_2, \text{/Ans/}*) \\ Q_2 & \leftarrow & (Q_1 \cap Q_2, \text{/Ans/}*) \\ Q_2 & \leftarrow & (Q_2 -_{\preceq} Q_1, \text{/Ans/}*) \end{array}$$

If $n < m$, $Q_1 \cap_{\prec} Q_2$, $Q_1 \cap Q_2$, and $Q_2 \cap_{\succ} Q_1$ are empty, and the three queries below constitute a minimal view:

$$\begin{array}{rl} Q_1 -_{\preceq} Q_2 : & /p_1^1/\ldots/p_1^n - /p_2^1/\ldots/p_2^n[p_2^{n+1}/\ldots/p_2^m] \\ Q_1 \cap_{\succ} Q_2 : & /p_1^1/\ldots/p_1^n \cap /p_2^1/\ldots/p_2^n[p_2^{n+1}/\ldots/p_2^m] \\ Q_2 -_{\preceq} Q_1 : & /p_2^1/\ldots/p_2^m - /p_1^1/\ldots/p_1^n//* \end{array}$$

and we can extract the final answers as follows:

$$\begin{array}{lcl} Q_1 & \leftarrow & (Q_1 -_{\preceq} Q_2, \text{/Ans/}*) \\ Q_1 & \leftarrow & (Q_1 \cap_{\succ} Q_2, \text{/Ans/}*) \\ Q_2 & \leftarrow & (Q_1 \cap_{\succ} Q_2, \text{/Ans/}*/p_2^{n+1}/\ldots/p_2^m) \\ Q_2 & \leftarrow & (Q_2 -_{\preceq} Q_1, \text{/Ans/}*) \end{array}$$

The third line above corresponds to the extraction of $Q_2 \cap_{\prec} Q_1$ from $Q_1 \cap_{\succ} Q_2$. This solution is directly following the observation explained above, but we can slightly simplify the view set above to the one shown below:

$$\begin{array}{rl} Q_{1-2} : & /p_1^1/\ldots/p_1^n - /p_2^1/\ldots/p_2^n \\ Q_{1\cap 2} : & /p_1^1/\ldots/p_1^n \cap /p_2^1/\ldots/p_2^n \\ Q_2 -_{\preceq} Q_1 : & /p_2^1/\ldots/p_2^m - /p_1^1/\ldots/p_1^n//* \end{array}$$

From this view set, we can extract the answer to Q_1 and Q_2 in exactly the same procedure as above with substituting $Q_1 -_{\preceq} Q_2$ and $Q_1 \cap_{\succ} Q_2$ with Q_{1-2} and $Q_{1\cap 2}$, respectively.
Proof: It is easy to show $(Q_1 -_{\preceq} Q_2) \cup (Q_1 \cap_{\succ} Q_2) = Q_{1-2} \cup Q_{1\cap 2}$. It is also easy to show that the result of both $(Q_1 \cap_{\succ} Q_2, \text{/Ans/}p_2^n/\ldots/p_2^m)$ and $(Q_{1\cap 2}, \text{/Ans/}p_2^n/\ldots/p_2^m)$ are equal to the answer to $Q_1 // * \cap Q_2$. □

For example, let Q_1 be /a/b and Q_2 be /a/∗/c. Then, $Q_1 -_{\preceq} Q_2$ is /a/b − /a/∗[c], and $Q_1 \cap_{\succ} Q_2$ is /a/b ∩ /a/∗[c], but we can also extract the answers to Q_1 and Q_2 from /a/b − /a/∗ and /a/b ∩ /a/∗ in the same procedure.

5.2 Algorithm

In the previous subsection, we explained the intuition behind our algorithm, and showed a simple algorithm for two queries. In this subsection, we show a complete algorithm for an arbitrary number of non-recursive union-free queries. In the simple algorithm, we computed $-_{\preceq}$ and $\cap_{\succ}$ of queries by using − and ∩ constructs for queries, and it may create unnecessary views which are always empty. On the other hand, the algorithm shown in this subsection produces simpler queries that do not include − and ∩, and does not produce unnecessary empty views.

The main part of the algorithm is translation of queries into automata, and construction of their product automaton. When given a set of non-recursive union-free XPath queries, we first translate them to a deterministic finite automata on a alphabet of symbols sym defined as below:

$$sym ::= a \mid \overline{\{a_1, \ldots, a_n\}} \mid * \mid sym[p] \mid sym\overline{[p]}$$

$$\begin{tikzcd} Q_1 : s_1^0 \arrow[r, "a"] & s_1^1 \arrow[r, "b"] & e_1 \end{tikzcd}$$
$$\begin{tikzcd} Q_2 : s_2^0 \arrow[r, "a"] & s_2^1 \arrow[r, "\overline{\{c\}}[c]"] & e_2 \end{tikzcd}$$
$$\begin{tikzcd} Q_3 : s_3^0 \arrow[r, "a"] & s_3^1 \arrow[r, "*"] & s_3^2 \arrow[r, "c"] & e_3 \end{tikzcd}$$

Figure 1: Automata for Queries Q_1, Q_2, Q_3

$$\begin{tikzcd} Q_2 : s_2^0 \arrow[r, "a"] \arrow[d, "\overline{\{a\}}"'] & s_2^1 \arrow[r, "\overline{\{c\}}[c]"] \arrow[dl, "c"'] \arrow[dl, bend left, "\overline{\{c\}}\overline{[c]}"] & e_2 \\ f_2 \arrow[loop left, "*"] & & \end{tikzcd}$$

Figure 2: Automaton for Q_2 with a fail state

where a or $a_1, \ldots, a_n$ are any label, and p is a relative location path defined before.

Because input queries do not include //, a query is translated into an automaton of the form of a simple sequence. For example, suppose we have the following set of queries: Q_1: /a/b, Q_2: /a/$\overline{\{c\}}$[c], Q_3: /a/∗/c. Those queries are translated into the three automata shown in Figure 1.

Then, we explicitly add a "fail state" to each automaton. For example, Figure 2 shows the automaton for Q_2 with the fail state. To add fail states, we need to compute the complementation of symbols. Complementation of symbols sym, denoted by $(sym)^-$, is defined by the following rules corresponding to the syntax definition of sym above:

$$\begin{array}{rcl} (a)^- & = & \{\overline{\{a\}}\} \\ (\overline{\{a_1, \ldots, a_n\}})^- & = & \{a_1, \ldots, a_n\} \\ (*)^- & = & \emptyset \\ (sym[p])^- & = & (sym)^- \cup \{sym\overline{[p]}\} \end{array}$$

Notice that the complementation of a symbol is represented by a set of symbols. Because of that, we may need many transition rules from each state to the fail state.

Then, we construct the product of all those automata in the standard way. The only difference from the standard product construction is that we need to compute the intersection and the difference between symbols. Intersection of two symbols, $\cap(sym, sym)$ is also defined by the rules corresponding to the syntax definition above. Here, we list only part of the rules:

$$\begin{array}{rcll} \cap(a, \overline{\{a_1, \ldots\}}) & = & a & \text{if } a \notin \{a_1, \ldots\} \\ & & \textit{undefined} & \text{otherwise} \\ \cap(\overline{\{a_1, \ldots\}}, \overline{\{b_1, \ldots\}}) & = & \overline{\{a_1, \ldots, b_1, \ldots\}} & \\ \cap(sym, sym[p]) & = & \cap(sym, sym)[p] & \\ \cap(sym, sym\overline{[p]}) & = & \cap(sym, sym)\overline{[p]} & \end{array}$$

Notice that intersection of two symbols can always be represented by a single symbol while complementation of a symbol can be a set of symbols. The difference of two symbols, $-(sym_1, sym_2)$, are computed by the rule below:

$$-(sym_1, sym_2) = \cap(sym_1, (sym_2)^-)$$

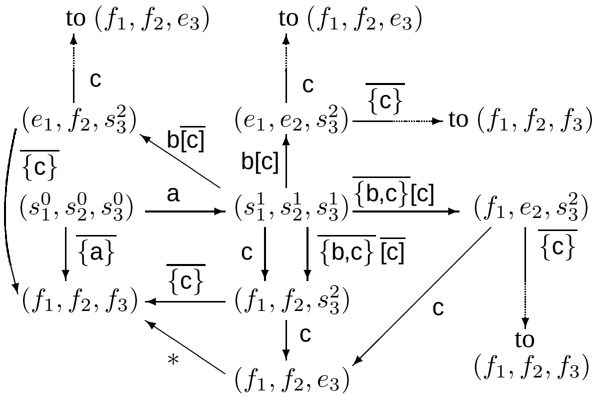

Figure 3: Product Automaton for Q_1, Q_2, Q_3

By using intersection and difference defined above, we construct a product of all the automata for the given queries. The product automaton for Q_1, Q_2, Q_3 is shown in Figure 3. Notice that automata for queries with the fail states forms DAG, and their product also forms DAG.

A product automaton constructed in this way may include paths that are never followed. For example, in the automaton shown in Figure 3, the transition with a label c from (e_1, f_2, s_3^2) to (f_1, f_2, e_3) is never followed because of b$\overline{[\text{c}]}$ on the only transition into (e_1, f_2, s_3^2). The path $\overline{\{\text{b,c}\}}\ \overline{[\text{c}]}$/c from (s_1^1, s_2^1, s_3^1) to (f_1, f_2, e_3) is also never followed (although each of $\overline{\{\text{b,c}\}}\ \overline{[\text{c}]}$ and c are used in other paths). There may also exist unsatisfiable symbols produced in the computation of intersection or difference. In addition, even the original queries submitted by the users may include some unsatisfiable conditions by mistake.

We can determine satisfiability of a path by testing the satisfiability of the set of predicates:

$$\{[pp_1], \ldots, [pp_n], \overline{[np_1]}, \ldots, \overline{[np_m]}, [p]\}$$

for each symbol $sym[pp_1] \ldots [pp_n]\overline{[np_1]} \ldots \overline{[np_m]}$ in the path and the suffix p of the path following that symbol (or $\{[pp_1], \ldots, [pp_n], \overline{[np_1]}, \ldots, \overline{[np_m]}\}$ if the suffix p does not exists). For example, the satisfiability of the path a[b]$\overline{[\text{b/c}]}$/b/c/d is determined by testing the satisfiability of $\{$[b], $\overline{[\text{b/c}]}$, [b/c/d]$\}$, which is unsatisfiable because of $\overline{[\text{b/c}]}$ and [b/c/d]. Therefore, this path is unsatisfiable.

We test satisfiability of a set of predicates as follows:

Proposition 6 *$\{[pp_1], \ldots, [pp_n], \overline{[np_1]}, \ldots, \overline{[np_m]}\}$ is not satisfiable iff some prefix of some pp_i is contained (in the ordinary sense of query containment) by some np_j.*

Proof: If no prefix of pp_i is contained by any of np_j, we can create a path that matches pp_i but not any of np_j. Then, an element with n children each of which satisfies one of $pp_1, \ldots, pp_n$ but not any np_j satisfies all the predicates. □

We can determine if any prefix of some pp_i is contained by some np_j by constructing a product automaton for /pp_i and /np_j as explained below.

By using the product automaton for $Q_1, \ldots, Q_n$, we can compute the following relations and operations on queries:

Proposition 7 *Q_i is contained by Q_j iff there is no satisfiable path from $(s_1, \ldots, s_n)$ to any states of the form $(\ldots, e_i, \ldots, s_j^k, \ldots)$ where $s_j^k \neq e_j$.*

Proposition 8 *Intersection of Q_i and Q_j, $Q_i \cap Q_j$, is a union of queries corresponding to all satisfiable paths from $(s_1, \ldots, s_n)$ to any states of the form $(\ldots, e_i, \ldots, e_j, \ldots)$.*

Proposition 9 *Difference of Q_i and Q_j, $Q_i - Q_j$, is a union of queries corresponding to all satisfiable paths from $(s_1, \ldots, s_n)$ to any states of the form $(\ldots, e_i, \ldots, s_j^k, \ldots)$ where $s_j^k \neq e_j$.*

Therefore, we can express the intersection and the difference of two queries without using $-$. In other words, the language without recursion and $-$ is closed under intersection and difference. The proofs of those propositions are easy and omitted here. The computation of containment is used to test the satisfiability of predicates as explained above, and the computation of intersection or difference will be used in the next section.

Now we show the algorithm.

```
Algorithm for Non-recursive Queries
Input: $n$ non-recursive queries $Q_1, \ldots, Q_n$.
Output: a set of queries $\{V_1, \ldots, V_m\}$ corresponding to the minimal view set, and a list of triplets $Q_i \leftarrow (V_j, q_i^j)$ showing how to extract answers to $Q_1, \ldots, Q_n$ from them.
begin
  1. Translate $Q_1, \ldots, Q_n$ into automata, add fail states explicitly, and construct a product automaton.
  2. For each satisfiable path $X$ from $(s_1, \ldots, s_n)$ to a state $T$ of the form $(\ldots, e_{i_1}, \ldots, e_{i_2}, \ldots, e_{i_a}, \ldots)$ that does not go through any other states of the form $(\ldots, e_j, \ldots)$:
     (a) add X to the view set, and add $Q_i \leftarrow (X, \text{/Ans/}*)$ to the triplet list for each $i \in \{i_1, \ldots, i_a\}$.
     (b) for each path $Y$ from the state $T$ to any state of the form $(\ldots, e_j, \ldots)$, if $X/Y$ is satisfiable, add a triplet $Q_j \leftarrow (X, \text{/Ans/}*/Y)$ to the list.
end □
```

For example, from the product automaton shown in Figure 3, the algorithm above produces a view set:

$$\{\text{/a/b}\overline{[\text{c}]},\ \text{/a/b[c]},\ \text{/a/}\overline{\{\text{b,c}\}}\text{[c]},\ \text{/a/c/c}\}$$

and the following triplets:

$$\begin{array}{ll}
Q_1 \leftarrow (\text{/a/b}\overline{[\text{c}]}, \text{/Ans/}*) & Q_1 \leftarrow (\text{/a/b[c]}, \text{/Ans/}*) \\
Q_2 \leftarrow (\text{/a/b[c]}, \text{/Ans/}*) & Q_2 \leftarrow (\text{/a/}\overline{\{\text{b,c}\}}\text{[c]}, \text{/Ans/}*) \\
Q_3 \leftarrow (\text{/a/c/c}, \text{/Ans/}*) & Q_3 \leftarrow (\text{/a/b[c]}, \text{/Ans/}*\text{/c}) \\
Q_3 \leftarrow (\text{/a/}\overline{\{\text{b,c}\}}\text{[c]}, \text{/Ans/}*\text{/c}) &
\end{array}$$

Please examine that we can correctly extract the answers to Q_1, Q_2, Q_3 by these procedures. Notice that the view set includes /a/c/c instead of /a/c/c$\cup$/a/$\overline{\{\text{b,c}\}}\ \overline{[\text{c}]}$/c because /a/$\overline{\{\text{b,c}\}}\ \overline{[\text{c}]}$/c is unsatisfiable. Similarly, $Q_3 \leftarrow$ (/a/b$\overline{[\text{c}]}$, /Ans/$*$/c) was not included in the triplet list because /a/b$\overline{[\text{c}]}$/c is unsatisfiable.

Theorem 1 *The algorithm above is correct.*

Proof Outline: Each query added to the view set in the step 2(a) in the algorithm above corresponds to a simplified version (explained at the end of the previous subsection) of $Q_{i_1} \cap \ldots \cap Q_{i_a} \cap_{\succ} Q_{j_1} \ldots \cap_{\succ} Q_{j_b} -_{\preceq} Q_{k_1} \ldots -_{\preceq} Q_{k_c}$ for some disjoint sets $\{j_1, \ldots, j_b\}$ and $\{k_1, \ldots, k_c\}$ s.t. $\{i_1, \ldots, i_a, j_1, \ldots, j_b, k_1, \ldots, k_c\} = \{1, \ldots, n\}$. From this view, we can extract part of answers to $Q_{i_1}, \ldots, Q_{i_a}$ by /Ans/$*$, and part of answers to $Q_{j_1}, \ldots, Q_{j_b}$ by /Ans/$*$/Y. In addition, the queries added to the view set do not include duplication under $\cap$ and even under $\cap_{\prec}$ or $\cap_{\succ}$. □

It is also easy to prove the following proposition on the number of queries to be evaluated on servers and on clients.

Theorem 2 *When given non-recursive union-free queries $Q_1, \ldots, Q_n$, we need to submit up to $2^n - 1$ queries to the server, and need up to $n * 2^{n-1}$ queries on the client.*

Proof: As an upper bound, the number of views cannot be larger than $2^n - 1$ because there cannot be larger number of the states of the form $(\ldots, e_{i_1}, \ldots, e_{i_2}, \ldots, e_{i_a}, \ldots)$ in the product automaton. We need to extract the answer to Q_i from up to $(2^x - 1) + 2^y$ views where x is the number of queries which are shorter than Q_i and y is the number of queries which have the same length as Q_i. It takes its maximum value 2^{n-1} when $\{x, y\} = \{n-1, 0\}$. Therefore, $n * 2^{n-1}$ is an upper bound for the number of queries evaluated on the client. As the lower bound, we actually need $2^n - 1$ views and $n * 2^{n-1}$ client queries if $Q_1, \ldots, Q_n$ are /a/b[c_1]/d, ..., /a/b[c_n]/d for some distinct $c_1, \ldots, c_n$. □

If we want to minimize the number of queries in the view set, we can merge two views V_1 and V_2 *iff* the intersection of two symbols on the transitions to the states corresponding to V_1 and V_2 is undefined or unsatisfiable. This is because only information that can be used to distinguish elements in the answer to $V_1 \cup V_2$ is the information represented by those two symbols. If they have intersection, and if the answer to $V_1 \cup V_2$ includes some element that matches that intersection, we cannot tell whether that element was belonging to V_1 or V_2 (or both). In this paper, however, we do not discuss this issue in more detail as mentioned in Section 4.

6 Algorithm for One Recursive Query

In this section, we show an algorithm that computes a minimal view set that can answer one given recursive query. Even though we restrict the input language of our algorithms to a language without $\cup$ and $-$ operations, the output language of the algorithms for recursive queries includes them. This is partly because the language with recursion but without $-$ operations is not closed under difference or complementation. For example, //a$-$//a//a cannot be expressed without $-$. (In XPath standard, it can be expressed by using an "ancestor axis," which we do not explain here. In this paper, we assume a language without an ancestor axis.) As shown in [6], most XPath fragments used in recent researches are closed under intersection but not closed under difference or complementation.

Suppose we are given a recursive query of the form:

$$Q: /p_1//p_2//\ldots//p_n \qquad \text{or} \qquad Q: //p_1//p_2//\ldots//p_n$$

where $p_1, \ldots, p_n$ are relative location paths that do not include //. Because whether the query starts with / or // does not matter in the following discussion, here we assume /.

As shown in the examples in Section 3, the redundancy in the answer to this query occurs in two ways:

- there are elements that match $/p_1//\ldots//p_n//p_n$
- there are elements that match $/p_1//\ldots//p_n/p$ where p is some suffix of p_n s.t. the remaining prefix of p_n matches the suffix of p_n. Please refer to the example of //a/b/a/b in Section 3.

If we have only the former kind of redundancy, we can simply submit a view query:

$$(/p_1//\ldots//p_n) - (/p_1//\ldots//p_n//*)$$

and produce the final answer by applying /Ans/$*$ and /Ans//p_n to the view. To also remove the latter kind of redundancy, we consider a set of relative location paths:

$$S = \{*/p_n^{(1,k-1)},\ */*/p_n^{(1,k-2)},\ \ldots,\ */\ldots/*/p_n^{(1,2)}\}$$

where k is the length of p_n, and $p_n^{(i,j)}$ is the subsequence of p_n from the position i to the position j. Then, the algorithm computes the following views:

$$V(T): (/p_1//\ldots//(p_n \cap \bigcap_{p \in T} p - \bigcup_{p \in S-T} p)) - /p_1//\ldots//p_n//*$$

for every $T \subseteq S$. Here, we use ordinary $\cap$ and $-$ because p_n and every $p \in S$ have the same length k, that is, their answers cannot be subelements of other answers. The algorithm computes $\cap$ and $-$ of paths by using the product automaton explained in the previous section. If the result of $p_n \cap \bigcap p - \bigcup p$ is empty for some T, $V(T)$ is discarded. Then, for each survived view $V(T)$, the algorithm produces the following triplets:

$$(Q, V(T), /\text{Ans}/*)$$
$$(Q, V(T), /\text{Ans}//p_n)$$
$$(Q, V(T), /\text{Ans}/*/p_n^{(i+1,k)}) \text{ for each } */\ldots/*/p_n^{(1,i)} \in T$$

When the length of p_n is 1, the first one can be omitted because the second one contains the first one.

For example, suppose we are given a query $Q =$ /a//b/c/$*$/$\overline{\{\text{d}\}}$. Then $S = \{p'_3\colon */\text{b}/\text{c}/*,\ p'_2\colon */*/\text{b}/\text{c}\}$, and the algorithm examines the following four views:

$$V_1: /\text{a}//\ (\text{b}/\text{c}/*/\overline{\{\text{d}\}}\ \cap\ p'_3\ \cap\ p'_2) - /\text{a}//\text{b}/\text{c}/*/\overline{\{\text{d}\}}//*$$
$$V_2: /\text{a}//\ (\text{b}/\text{c}/*/\overline{\{\text{d}\}}\ \cap\ p'_3\ -\ p'_2) - /\text{a}//\text{b}/\text{c}/*/\overline{\{\text{d}\}}//*$$
$$V_3: /\text{a}//\ (\text{b}/\text{c}/*/\overline{\{\text{d}\}}\ \cap\ p'_2\ -\ p'_3) - /\text{a}//\text{b}/\text{c}/*/\overline{\{\text{d}\}}//*$$
$$V_4: /\text{a}//\ (\text{b}/\text{c}/*/\overline{\{\text{d}\}}\ -\ p'_3\ -\ p'_2) - /\text{a}//\text{b}/\text{c}/*/\overline{\{\text{d}\}}//*$$

By computing $\cup$ and $-$ with the product automaton, the algorithm find V_1 and V_2 are empty, and finally produces a view set consisting of the following two views:

$$\begin{array}{l} V_3 : \text{/a//b/c/b/c} - \text{/a//b/c/}*\text{/}\overline{\{\text{d}\}}\text{//}* \\ V_4 : (\text{/a//b/c/}\overline{\{\text{b}\}}\text{/}\overline{\{\text{d}\}} \cup \text{/a//b/c/b/}\overline{\{\text{c,d}\}}) - \text{/a//b/c/}*\text{/}\overline{\{\text{d}\}}\text{//}* \end{array}$$

The algorithm also produces the following triplets:

$$\begin{array}{lcl} Q & \leftarrow & (V_3, \text{/Ans/}*) \\ Q & \leftarrow & (V_3, \text{/Ans//b/c/}*\text{/}\overline{\{\text{d}\}}) \\ Q & \leftarrow & (V_3, \text{/Ans/}*\text{/}*\text{/}\overline{\{\text{d}\}}) \\ Q & \leftarrow & (V_4, \text{/Ans/}*) \\ Q & \leftarrow & (V_4, \text{/Ans//b/c/}*\text{/}\overline{\{\text{d}\}}) \end{array}$$

Theorem 3 *The algorithm above is correct.*

Proof Outline: The view set above does not include self-redundancy because of $-(\text{//}p_1\text{//}\ldots\text{//}p_n\text{//}*)$ at the tail of each view. Next, we show that the triplets above extract all the answers. Answers that appear as the first answers in the paths from the root are extracted by /Ans/$*$. We call those answers "top-most answers". Answers that appear as children of top-most answers are extracted by /Ans/$*$/$p_n^{(k,k)}$, answers that appear as grandchildren are extracted by /Ans/$*$/$*$/$p_n^{(k-1,k)}$, $\ldots$, and so on, and finally answers that appear k or more levels deeper than the top-most answers are extracted by /Ans//p_n. □

Theorem 4 *When given one recursive query, we need up to 2^{k-2} (or 1 when $k = 1$) queries to the server where k is the length of the longest non-recursive suffix of the query (i.e., p_n above). On the client, we need to evaluate up to $2^{k-1} + (k-2) * 2^{k-3}$ (or 2 when $k = 1$) queries.*

Proof: The algorithm above creates up to 2^{k-2} views. On the client, we may need to execute two queries /Ans/$*$ and /Ans//p_n for all of 2^{k-2} views, which amounts to 2^{k-1} queries, and also need to evaluate each /Ans/$*$/$p_n^{(i,k)}$ ($3 \leq i \leq k$) on up to 2^{k-3} views, and it amounts to $(k-2) * 2^{k-3}$. As the lower bound, we actually need those number of views and client queries when we have a query of the form /a//$\overline{\{b_1\}}$/$\ldots$/$\overline{\{b_k\}}$ for some distinct $b_1, \ldots, b_k$. □

7 Algorithm for Recursive Queries

By combining the intuition shown in 5.1 and the algorithm in the previous section, we develop an algorithm that computes a minimal view set in general case, i.e., when given the following set of recursive queries:

$$\begin{array}{c} Q_1 : /_1^1\, p_1^1\, /_1^2\, p_1^2 \ldots /_1^{l_1}\, p_1^{l_1} \\ \vdots \\ Q_n : /_n^1\, p_n^1\, /_n^2\, p_n^2 \ldots /_n^{l_n}\, p_n^{l_n} \end{array}$$

where p_i^j is an expression that includes neither / nor //, and each $/_i^j$ represents either / or //. We define prefix paths $pp_i^j (1 \leq i \leq n, 0 \leq j \leq l_i - 1)$ as follows :

$$pp_i^j \equiv \begin{cases} /_i^1 p_i^1 \ldots /_i^j p_i^j & \text{if } /_i^{j+1} = \text{/} \\ (/_i^1 p_i^1 \ldots /_i^j p_i^j) \cup (/_i^1 p_i^1 \ldots /_i^j p_i^j\text{//}*) & \text{if } /_i^{j+1} = \text{//} \\ \emptyset & \text{if } j = 0,\ /_i^1 = \text{/} \\ \text{//}* & \text{if } j = 0,\ /_i^1 = \text{//} \end{cases}$$

where $\emptyset$ is the empty path that matches no elements.

Then, we create views defined as below for any S, T s.t. $S \subseteq \{1, \ldots, n\}$, $S \neq \emptyset$, $T \subseteq \{(i, j) \mid 1 \leq i \leq n, 0 \leq j \leq l_i - 1\}$:

$$(\bigcap_{i \in S} Q_i - \bigcup_{i \notin S} Q_i) \cap (\bigcap_{(i,j) \in T} pp_i^j - \bigcup_{(i,j) \notin T} pp_i^j) - \bigcup_{1 \leq i \leq n} Q_i\text{//}*$$

Let $V(S, T)$ denote a view defined with S, T. Then, for each $V(S, T)$, we produce the following triplets:

$$\begin{array}{ll} Q_i \leftarrow (V(S,T), \text{/Ans/}*) & \text{for } i \in S \\ Q_i \leftarrow (V(S,T), \text{/Ans/}*/_i^{j+1}\ p_i^{j+1} \ldots /_i^{l_i}\ p_i^{l_i}) & \text{for } (i, j) \in T \end{array}$$

For example, suppose we are given two queries:

$$\begin{cases} Q_1 : & \text{//a} \\ Q_2 : & \text{/b//}\overline{\{\text{c}\}} \end{cases}$$

Then, $pp_1^0 = \text{//}*$, $pp_2^0 = \emptyset$, and $pp_2^1 = \text{/b} \cup \text{/b//}*$ are defined for Q_1 and Q_2. We can consider three sets for S and eight sets for T. For T, however, we only need to consider those including pp_1^0 and not including pp_2^0 because views created by other T are empty. In addition, $\cap pp_1^0$ and $-pp_2^0$ in the view queries can be omitted because they do not change the semantics of the entire query. As a result, we create the following views:

$$\begin{array}{l} V_1 : (Q_1 \cap Q_2) \cap pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \\ V_2 : (Q_1 \cap Q_2) - pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \\ V_3 : (Q_1 - Q_2) \cap pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \\ V_4 : (Q_1 - Q_2) - pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \\ V_5 : (Q_2 - Q_1) \cap pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \\ V_6 : (Q_2 - Q_1) - pp_2^1 - (Q_1\text{//}* \cup Q_2\text{//}*) \end{array}$$

We also produce the following triplets:

$$\begin{array}{lcll} Q_1 & \leftarrow & (V_i, \text{//Ans/}*) & \text{where } i \in \{1, 2, 3, 4\} \\ Q_2 & \leftarrow & (V_i, \text{//Ans/}*) & \text{where } i \in \{1, 2, 5, 6\} \\ Q_1 & \leftarrow & (V_i, \text{//Ans/}*\text{//a}) & \text{where } i \in \{1, 2, 3, 4, 5, 6\} \\ Q_2 & \leftarrow & (V_i, \text{//Ans/}*\text{//}\overline{\{\text{c}\}}) & \text{where } i \in \{1, 3, 5\} \end{array}$$

Theorem 5 *The algorithm above is correct.*

Proof Outline: Because of $-\bigcup Q_i\text{//}*$ at the tail of every view query, this view set only includes top-most answers to $Q_1, \ldots, Q_n$. Each top-most answer e appear exactly once in the view set; e appears only in $V(S, T)$ s.t. $S = \{i \mid e \in Q_i(t)\}$ and $T = \{(i, j) \mid e \in pp_i^j(t)\}$ where t is the database tree. Therefore, the view set includes all the necessary elements without redundancy. From this view set, we can correctly extract all the answers. It is intuitively because what the algorithm does is to classify all the topmost answers based on how their subelements should be extracted as other answers. □

This algorithm may create many empty views. For example, $V_1, \ldots, V_6$ shown above can be simplified into the following queries:

$$\begin{array}{l} V_1 : \text{/b//a} - \text{//a//}* - \text{/b//}\overline{\{\text{c}\}}\text{//}* \\ V_4 : \text{//a} - \text{/b//}* - \text{//a//}* \\ V_5 : \text{/b//}\overline{\{\text{a,c}\}} - \text{//a//}* - \text{/b//}\overline{\{\text{c}\}}\text{//}* \\ V_2, V_3, V_6 : \emptyset \end{array}$$

Therefore, V_2, V_3, V_6 can be discarded. For such query simplification and empty view elimination, we need to solve the containment problem of XPath queries including //. We could use the techniques shown in the past researches, such as [3, 23], but that is out of the scope of this paper.

Theorem 6 *When given n recursive queries whose total length is l, we need up to $(2^n - 1) * 2^{l-n}$ queries to the server, and we need up to $n * 2^{n-1} * 2^{l-n} + (l-n) * (2^n - 1) * 2^{l-n-1} + n * (2^n - 1) * 2^{l-n}$ queries on the client.*

Proof Outline: We have $2^n - 1$ different S and 2^l different T, but as explained above, for each pp_i^0, we need to consider only T including or not including pp_i^0, thus only 2^{l-n} different T. We need $2^{n-1} * 2^{l-n}$ queries of the form /Ans/$*$ for each Q_i, $(2^n - 1) * 2^{l-n-1}$ queries for each $pp_i^j (j \leq 1)$, and $(2^n - 1) * 2^{l-n}$ queries for each pp_i^0. As the lower bound, we actually need those number of views and client queries when we have queries $Q_1, \ldots, Q_n$ of the form Q_i : //$\overline{\{a_i^1\}}$/…/$\overline{\{a_i^{l_i}\}}$ for some distinct a_i^j. □

8 Related Work

There have been a large number of researches on the view selection problem [18]. The main goal of the traditional view selection problem is to choose a set of views that minimizes the cost of answering queries within a limited resource for storing views, and also within a limited cost for maintaining them. On the other hand, the goal of our research is to minimize the size of the data sent between servers and clients over networks, which may actually increase computation costs both on servers and clients.

A similar idea of computing minimal views to reduce communication costs has been discussed in [11]. In that paper, the authors discuss the problem of minimal views in the context of relational databases, conjunctive queries, and the redundancy caused by join operations. In this paper, we discuss the minimal view problem in the context of XML (or any data with nested structure), XPath queries (or any language for nested data structure), and the redundancy caused by the nested structure in the data.

In the context of client-server database architecture, the concepts of semantic caching and remainder queries have been proposed in [14], and they have also been studied in the context of XML data in [10]. In semantic caching, the client caches the answers to previous queries together with the query expressions. When a user on the client issues a new query whose answer partially overlaps with answers to some previous queries, the client computes and submits a "remainder query" that only retrieves data that are not available in the cached answers. In general, however, some context information may not be available in the cached answers nor in their query expressions, and therefore, it is not always possible to correctly extract part of answers to new queries from the cached data only by looking at the expressions and the answers of the cached queries. For example if we submit Q_9 in Section 3 first, and submit Q_{10} later, we cannot extract the answers to Q_{10} from the cached answer to Q_9. On the other hand, in our problem setting, first we are given a set of queries. For that, we can divide given queries into smaller queries before submitting them so that we can extract answers to overlapping queries. In addition, [14] and [10] does not consider the duplication caused by answers appearing as substructure of other answers.

The optimization of communication costs in query processing over a network has also been studied in the context of distributed databases [4], where distributed data servers cooperate. In this paper, however, we assumed an environment where all clients can do is to submit queries, and they cannot use special encodings or protocols.

The view minimization problem has also been studied in [20]. Their goal is to minimize given views without losing the power to answer queries, while our goal is to compute a minimal view that can answer a given set of queries.

There also have been researches on answering queries on tree or graph structured data using views [7, 15]. Their goal is, however, to answer queries with a given view set, not to compute minimal view set for a given set of queries.

9 Discussion and Conclusion

In this paper, we studied a problem in XML database systems on networks, which has recently become very important both in the academy and in the industry. The problem is the redundancy in the answers to XPath queries sent over the network, which wastes network resources. A similar problem can occur in other data models and query languages, but this problem occurs especially frequently in the context of XML and XPath. Even when a user submits a single, quite ordinary XPath query, the answer to it may include significant redundancy. This problem comes from the characteristics of XML and XPath: the data have a nested structure and the language retrieves substructures appearing at arbitrary levels. Therefore, although this paper discussed the problem in the context of XML and XPath, similar problems occur in any nested data structure and query languages that retrieves substructure at arbitrary level.

To solve this problem, we proposed the minimal view approach. Given a set of queries, we compute a minimal view set that can answer all the given queries, and submit the queries asking for that view set to the database. Then, the database sends that view set to the client, and the client uses it to produce the answers to the original queries. We showed algorithms that compute such a minimal view set.

Because view sets we compute are minimal in size, we can optimize communication costs between database servers and the clients.

One problem in this approach is that the queries for a minimal view set are usually more complex than its original queries, and it may increase the computation cost on the server. To verify that this problem is not too serious for our approach to be practical, we conducted experiments to examine how much our approach improves communication costs for practical queries, and how it affects computation costs on the servers. Here, due to space limitations, we only briefly summarize the result of our experiments. The detail of the experiments will be reported in another publication.

As test data, we generated 233MB of artificial auction data by XMark [26]. We ran experiments in two settings. First, we stored XML data in a plain file, and evaluated XPath using a DOM-based in-memory XPath processor Xalan [27]. Second, we stored the data in a RDBMS, Oracle 9i, using a standard relational encoding scheme of XML used in many researches, such as [16], and evaluated XPath by transforming them into SQL. We tested various practical queries, and for non-recursive queries, we could even reduce the computation cost on the server in many cases. For example, we tested the queries below:

Q_1 : /site/region/namerica/item
Q_2 : /site/region/europe/item
Q_3 : /site/region/$*$/item/description

which asks for complete information on auction items in North America and Europe, and also asks for descriptions of auction items in any region. Our algorithm computes the view set consisting of Q_1 and Q_2 above, and q_3 below:

q_3 : /site/region/$\overline{\{\text{namerica,europe}\}}$/item/description

Then, the total size of the query results, i.e., the size of the data to be sent over the network was reduced by more than 60%. It is not surprising, but a more surprising result is even the computation cost was reduced slightly in the DOM setting, and by more than 25% in the relational encoding setting. This is because the evaluation cost includes some factors which are proportional to the answer size, and view queries are more complicated but have smaller answers.

For recursive queries, if we evaluate $-Q_i$//* directly, its computation cost was very high. However, by expanding // into a union of many queries, we could reduce the computation cost in many cases. The detail of such optimization of the queries produced by our algorithm shown in this paper is an important future work.

It is also interesting to investigate the interaction between our approach and the compression approach, which compresses the data before sending, and decompresses it on the client. Because compression removes redundancy, it may offset the difference of the size of the original answers and our minimal views. It is another important future work.

References

[1] M. Cherniack, S. B. Zdonik, M. H. Nodine: To Form a More Perfect Union (Intersection, Difference). In *DBPL*, 1995

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *VLDB*, pp. 53–64, 2000.

[3] S. Amer-Yahia, *et al*. Minimization of tree pattern queries. In *SIGMOD*, pp. 497–508, 2001.

[4] P. M. G. Apers. Data allocation in distributed database systems. *TODS*, 13(3):263–304, 1988.

[5] C. Barton, *et al*. Streaming XPath processing with forward and backward axes. In *ICDE*, pp. 455–466, 2003.

[6] M. Benedikt, W. Fan, G. M. Kuper. Structural properties of XPath fragments. In *ICDT*, pp. 79–95, 2003.

[7] D. Calvanese, *et al*. Answering regular path queries using views. In *ICDE*, pp. 389–398, 2000.

[8] C.-Y. Chan, *et al*. Efficient filtering of XML documents with XPath expressions. In *ICDE*, pp. 235–244, 2002.

[9] J. Chen, *et al*. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pp. 379–390, 2000.

[10] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery answering system. In *WebDB*, pp. 31–36, 2002

[11] R. Chirkova and C. Li. Materializing views with minimal size to answer queries. In *PODS*, pp. 38–48, 2003.

[12] J. Clark and S. DeRose, eds. *XML Path Language (XPath) Version 1.0 – W3C Recommendation*, 1999.

[13] J. Clark and S. DeRose, eds. *XML Path Language (XPath) Version 2.0 – W3C Working Draft*, 2003.

[14] S. Dar, *et al*. Semantic Data Caching and Replacement. In *VLDB*, pp. 330–341, 1996.

[15] G. Grahne and A. Thomo. Query containment and rewriting using views for regular path queries under constraints. In *PODS*, pp. 111–122, 2003.

[16] T. Grust. Accelerating XPath location steps. In *SIGMOD*, pp. 109–120, 2002.

[17] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, pp. 419–430, 2003.

[18] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[19] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

[20] C. Li, M. Bawa, J. D. Ullman. Minimizing view sets without losing query-answering power. In *ICDT*, pp. 99–113, 2001.

[21] L. Liu, C. Pu, W. Tang. Continual queries for internet scale event-drive information delivery. *TKDE*, 11(4):610–628, 1999.

[22] B. Ludäscher, P. Mukhopadhyay, Y. Papakonstantinou. A transducer-based XML query processor. In *VLDB*, pp. 227–238, 2002.

[23] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, pp. 65–76, 2002.

[24] D. Olteanu, *et al*. XPath: looking forward. In *XMLDM*, pp. 109–127, 2002.

[25] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD*, pp. 431–442, 2003.

[26] A. Schmidt, *et al*. XMark: A benchmark for XML data management. In *VLDB*, pp. 974–985, 2002.

[27] Xalan. `http://xml.apache.org/xalan-j/`.