The Framework of an XML Semantic Caching System

Wanhong Xu

Center for Computational Genomics
Department of Electrical Engineering and Computer Science
Case Western Reserve University, Cleveland, OH
Wanhong.Xu@case.edu

ABSTRACT

As a simple XML query language but with enough expressive power, XPath has become very popular. To expedite evaluation of XPath queries, we consider the problem of caching results of popular XPath queries to answer queries.

Existing semantic caching systems can answer queries that have been already cached, but can't combine cached results of multiple XPath queries to answer new queries. In this paper, we describe the architecture of a new semantic caching system, and mainly introduce the novel framework of this system. We show that our framework can represent cached XML data by a set of XPath queries, and the cached data can be used to answer new queries that may not be cached. We also consider incrementally maintaining the cached XML data in our system. The results suggest that our caching system is practical and has better answerability.

1. INTRODUCTION

Recently, more and more data are represented and exchanged as XML documents over Internet. XPath [9], recommended by W3C, is a simple but popular language to navigate XML documents and extract information from them, and to be used as sub-languages of other XML query languages such as XQuery [6].

There has been a lot of work to speedup evaluation of XPath queries, including, index techniques [8], structural join algorithms [2] and minimization of XPath queries [3, 15, 11]. Recently, the problem of answering queries using cached results in XML world has begun to attract more attention since it has been proven that the caching technique can improve performance significantly in traditional client-server databases, distributed databases and Web-based information systems. This problem has been discussed for query optimization [5] and semantic caching [7, 1, 12, 16].

We begin by giving some examples in the semantic caching scenario to describe motivation of studying this problem.

Copyright is held by the author/owner.
Eighth International Workshop on the Web and Databases (WebDB 2005), June 16-17, 2005, Baltimore, Maryland.

Motivation Examples: Consider the following XML document t stored in an XML server, which partially describes enzyme information of a biological pathway:

```
<Pathway name = "PA1">
  <Reaction name = "RE1">
    <Enzymes>
      <Protein name = "PR1" EC# ="1.0.0.1"/>
      <RNA name = "RN1"/>
    </Enzymes>
  </Reaction>
  <Reaction name = "RE2">
    <RNA name = "RN2">
  </Reaction>
</Pathway>
```

Let's assume that a client issues to the server an XPath query v :

$$//RNA$$

which retrieves all **RNA** elements. Suppose the client caches the result of v. When the client issues another XPath query p :

$$/Reaction/RNA$$

which retrieves **RNA** subelements of all **Reaction** elements. We know that the result of p is included in the cached result, i.e., the result of v. But, we can't compute p's result by using the cached result because we don't know which **RNA** elements in cached result belong to the result of p. In fact, the cached result of v can only be used to answer the same query as v. However, if we can store extra information, for example, the paths from the root of XML document t to each RNA subelement, then the cached result of v can be used to answer p. The cached result with path information, denoted as t', is given as follows:

```
<Pathway>
  <Reaction>
    <Enzymes>
      <RNA name = "RN1"/>
    </Enzymes>
  </Reaction>
  <Reaction>
    <RNA name = "RN2">
  </Reaction>
</Pathway>
```

t' is still an XML document. We can get the same result of evaluating p over t' as that of evaluating p over t. Hence,

storing extra information can improve the answerability of cached results.

In this paper, we introduce a framework to represent the cached XML document, discuss how to decide that the cached information is enough to answer a query, and how to incrementally maintain the cached XML document.

2. PRELIMINARIES

2.1 Trees and Tree Patterns

Generally, an XML database consists of a set of XML documents. We model the whole XML database as an unordered rooted node-labelled tree (called **XML tree**) over an infinite alphabet Σ (A virtual root node might be introduced to connect all XML documents if necessary). In this XML tree, each internal node's label corresponds to an XML element or attribute name, and each leaf node's label corresponds to a data value. In addition, we assume that each node has a unique node identifier. An XML tree is shown in Fig. 1 (a) as an instance. We let T_Σ be the set including all possible XML trees over Σ. Formally, we have:

Definition 2.1. *An XML database is a tree $t\langle V_t, E_t, r_t\rangle$ over Σ called **XML tree**, where*

- *V_t is the node set and E_t is the edge set;*
- *$r_t \in V_t$ is the root of t;*
- *Each node n in V_t has a label from Σ(denoted as $n.label$) and a unique node identifier (denoted as $n.id$).*

Given an XML tree $t\langle V_t, E_t, r_t\rangle$, the size of t is defined as the cardinality of V_t, and we also say that $t'\langle V_{t'}, E_{t'}, r_{t'}\rangle$ is a **subtree** of t if $V_{t'} \subseteq V_t$ and $E_{t'} = (V_{t'} \times V_{t'}) \cap E_t$. If t' includes the root of t, t' is also called as the **rooted subtree** of t.

In this paper, we discuss a fragment of XPath queries denoted as $XP^{\{/,//,*,[]\}}$, as in [13]. This fragment consists of label tests, child axes(/), descendant axes(//), branches([]) and wildcards(*). It can be recursively represented by the following grammar:

$$xp \rightarrow l | * | xp/xp | xp//xp | xp[xp]$$

where l is a node label from Σ. As said in [13], any XPath query from $XP^{\{/,//,*,[]\}}$ can be trivially represented as a labelled tree(called **tree pattern**) with the same semantics.

Definition 2.2. *A **tree pattern** p is a tree $\langle V_p, E_p, r_p, o_p\rangle$ over $\Sigma \cup \{$'*'$\}$, where V_p is the node set and E_p is the edge set, and:*

- *Each node n in V_p has a label from $\Sigma \cup \{$'*'$\}$, denoted as $n.label$;*
- *Each edge e in E_p has a label from {'/','//'}, denoted as $e.label$. The edge with label '/' is called child edge, otherwise called descendent edge;*

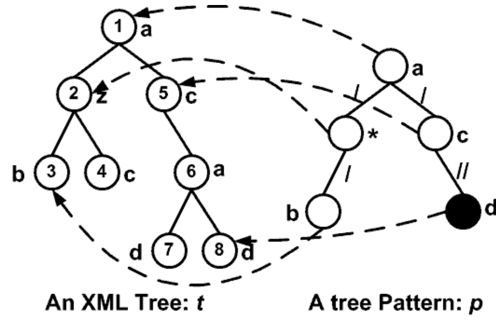

Figure 1: (a)An XML tree t; (b)A pattern p.

- *$r_p, o_p \in V_p$ are the root and output node of p respectively.*

For example, an XPath query $a[*/b]/c//d$ from $XP^{\{/,//,*,[]\}}$ is represented as a tree pattern shown in Fig. 1(b), where the dark node is the output node. Without loss of generality, we refer to tree patterns as patterns in the rest of this paper.

We now define an **embedding** (also called **pattern match**) from a pattern to an XML tree as follows:

Definition 2.3. *Given an XML tree $t\langle V_t, E_t, r_t\rangle$ and a pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an **embedding** from p to t is a function $e : V_p \rightarrow V_t$, with following properties:*

- *Root preserving: $e(r_p) = r_t$;*
- *Label preserving: $\forall n \in V_p$, if $n.label \neq$ '*', $n.label = e(n).label$;*
- *Structure preserving: $\forall e = (n_1, n_2) \in E_p$, if $e.label =$ '/', $e(n_2)$ is a child of $e(n_1)$ in t; otherwise, $e(n_2)$ is a descendent of $e(n_1)$ in t.*

The embedding maps the output node o_p of p to a node n in t. We say that the node n is the result of this embedding. As an example, dashed lines between Fig. 1(a) and (b) shows an embedding, and its result is the node with id = 8. Actually, there could be more than one embedding from p to t. We define the result of p over t, denoted as $p(t)$, as the union of results of all embeddings, i.e.,

$$\cup_{e \in EB}\{e(o_p)\}$$

where EB is the set including all embeddings from p to t.

For a given XML tree t, we also consider evaluating a set of patterns $S = \{p_1, p_2, ..., p_n\}$ over t. The result, denoted as $S(t)$, is the union of the result of evaluating each p_i in S over t, formally defined as:

$$S(t) = \cup_{p_i \in S} p_i(t)$$

2.2 Containment of Tree Patterns

For any two patterns p_1 and p_2, p_1 is said to be *contained* in p_2(denoted as $p_1 \sqsubseteq p_2$) iff $\forall t \in T_\Sigma\ p_1(t) \subseteq p_2(t)$. Similarly,

we also say that a pattern p is contained in a pattern set S(denoted as $p \sqsubseteq S$) iff $\forall t \in T_\Sigma\ p(t) \subseteq S(t)$, and a pattern set S_1 is contained in a pattern set S_2(denoted as $S_1 \sqsubseteq S_2$) iff $\forall t \in T_\Sigma\ S_1(t) \subseteq S_2(t)$.

It's easy to show that $S_1 \sqsubseteq S_2$ iff $\forall p_i \in S_1\ p_i \sqsubseteq S_2$. However, it's not always true that $p \sqsubseteq S$ implies $\exists p' \in S$ s.t. $p \sqsubseteq p'$.

3. SYSTEM OVERVIEW

In this section, we describe the architecture of our caching system, which is designed to work with XML web services and improve their performance.

Generally, web services are running at web servers and they act as bridges between clients and XML servers. When web services got service requests from clients, they would issue a series of XML queries to XML servers and return the corresponding results to clients.

The caching system runs as an independent application in the web server. Its architecture is shown in Fig. 2. This system intercepts all XML queries issued by web services, and tries to answer them by using local cached XML data instead of submitting them to the XML database server. It consists of several components. Next, We describe them one by one.

The *Cache* stores cached XML data with a semantic scheme. This semantic scheme consists of a set of patterns, and describes current cached XML data. The cached data is organized as an XML tree, which is a rooted subtree of the XML tree exported by the XML database server, more details given in Section 4.1.

When received an XML query, the *Query Manager* decides whether the cached XML tree can *totally answer* this query according to current semantic scheme or not, i.e., we can get the same result of evaluating this query over the cached XML tree as that of running it on the server, further discussed in Section 4.2. The *Index* built on the semantic scheme is used to speed-up the decision. If yes, the *Query Manager* will run this query against the local cached XML tree; otherwise submit it to the server.

The *Replacement Manager* employs replacement strategies (like **LRU**) to clear out less queried or expired data and incorporate new data. Specific details about how to incrementally maintain the cached XML tree will be given in Section 4.3.

4. FORMAL FRAMEWORK

We next introduce our formal framework and discuss how to represent, query and maintain the XML data cached in our caching system.

4.1 Representing the Cached XML Data

We discuss our representation of the cached XML information in this subsection. The main idea is that the cached XML information is represented by a set of patterns S, and this representation must satisfy two requirements: one is that the cached XML data must be a rooted sub-tree of the XML tree exported by the XML database server; the other

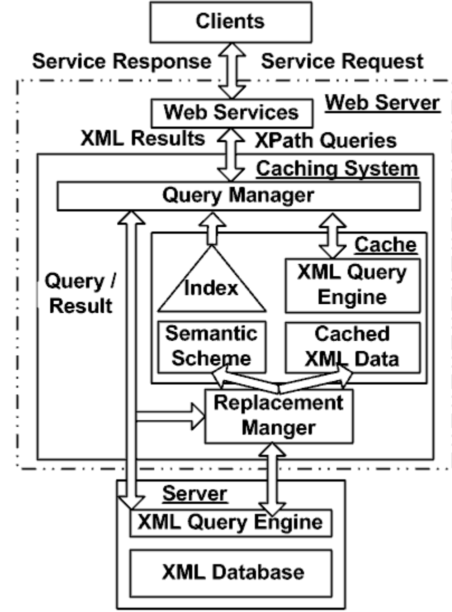

Figure 2: System Architecture

one is that the cached XML tree must ***totally answer*** any pattern in S, i.e, for any pattern $p \in S$, we can get the same result of evaluating this pattern p over the cached XML tree as that of evaluating p over the exported XML tree of the server.

We will describe this representation, satisfying the above two requirements, by defining the XML tree to be cached for a pattern set S. We formalize the representation for one pattern first, and then extend it to a pattern set later.

Before introducing the representation, we define the embedding node set of a pattern p over an exported XML tree t. This set includes all nodes in t mapped from nodes in p by an embedding between p and t. Hence, nodes in this set may be queried or accessed in the evaluation of p over t. Formally, we have:

Definition 4.1. *Given an exported XML tree t and a pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an **embedding node set** of p over t, denoted as $ENS(p,t)$, is $\cup_{e \in EB}(\cup_{v_i \in V_p}\{e(v_i).id\})$ where EB is the set including all possible embeddings from p to t and e is an embedding.*

Example 4.2. *In Fig. 3, there are only two embeddings from pattern p_1 shown in (a) to an XML tree t shown in (b). The dashed lines represent one embedding, and it maps nodes in p_1 to nodes with id = 1, 2 and 8 respectively in t. The dotted lines represent the other one, and it maps nodes in p_1 to the nodes with id = 1, 3 and 8 respectively in t. So, the embedding node set for p_1 is the union of nodes mapped by these two embeddings, i.e., $\{1,2,8\} \cup \{1,2,3\} = \{1,2,3,8\}$.*

We next generally define a materialized tree for any node set N over an exported XML tree t.

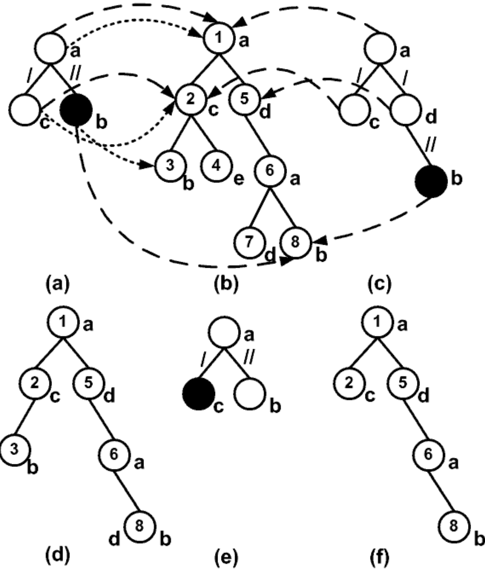

Figure 3: (a)Pattern p_1 (b)The exported XML tree t (c) Pattern p_2 (d) The XML tree t_{p_1} represented by p_1 (e)Pattern p_3 (f)The XML tree t_{p_2} represented by p_2

DEFINITION 4.3. *Given an exported XML tree $t\langle V_t, E_t, r_t\rangle$ and a set of nodes $N \subseteq V_t$, we say that a rooted subtree $t'\langle V_{t'}, E_{t'}, r_{t'}\rangle$ of t is a **materialized tree** for N over t if $N \subseteq V_{t'}$.*

However, there are maybe more than one materialized trees for a set of nodes N over an exported XML tree t. We are only interested in those with the minimum size. We say t' is the **minimal materialized tree**(***MMT***) for N over t if t' is a materialized tree for N over t and any materialized tree t'' for N over t has larger size than t'.

EXAMPLE 4.4. *In Fig. 3, let's assume that a node set $N = \{1, 2, 8\}$. Both XML trees shown in (d) and (f) respectively are materialized trees for N over the XML tree shown in (b). But, the XML tree shown in (f) is a minimal materialized tree of N, and the XML tree shown in (d) is not.*

Not only minimal materialized trees have the smallest sizes, but also they have very good properties shown in the following lemma.

LEMMA 4.5. *Given an exported XML tree t and a set of tree nodes N, t', which is a minimized materialized tree for N over t, has the following properties:*

- *For any materialized tree t'' for N over t, t' is a rooted subtree of t'';*
- *t' is unique, i.e., any other minimized materialized tree is identical to t'.*

We say that the XML tree represented by a pattern p over an exported XML tree t is the minimal materialized tree for the node set $ENS(p, t)$ over t, denoted as t_p.

Next, we generalize the above definitions to a pattern set $S = \{p_1, p_2, ..., p_n\}$. Given an exported XML tree t, we say that the embedding node set for S, denoted as $ENS(S, t)$, is the union of the embedding node set for each $p_i \in S$, i.e.,

$$\cup_{p_i \in S} ENS(p_i, t)$$

and the XML tree represented by this pattern set S, denoted as t_S, is the minimal materialized tree for the node set $ENS(S, t)$ over t.

Our representation satisfies the two requirements listed in the beginning of this subsection. The first one is obvious, and the second one is guaranteed by the following theorem.

THEOREM 1. *Given an exported XML tree t and a pattern set S, the XML tree t_S represented by S can totally answer any pattern p_i in S, i.e., $p_i(t) = p_i(t_S)$.*

4.2 Querying the Cached XML Tree

Given an XML tree t exported by the XML database server and a set of tree pattern S, we cache t_S in the web server. We want to use this t_S to answer patterns issued by clients. But, we need to assure that t_S can totally answer them, before evaluating them against t_S. From Theorem 1, we know that t_S can totally answer those patterns included in S. However, t_S can totally answer more patterns not only in S. This is the advantage of our framework. In this subsection, we will discuss the problem how to decide whether t_S can totally answer a pattern p(maybe not in S) or not. The basic idea is to check whether t_p (represented by p) is a rooted subtree of t_S or not. We have the following result:

LEMMA 4.6. *Given an exported XML tree t, a pattern p and a pattern set S, t_S can totally answer p if t_p is a rooted subtree of t_S.*

From the above lemma, our problem is reduced to decide whether or not t_p is a rooted subtree of t_S. We can further reduce our problem to decide whether $ENS(p, t) \subseteq ENS(S, t)$ or not in our next result.

LEMMA 4.7. *Given an exported XML tree $t\langle V_t, E_t, r_t\rangle$ and two node set $N(\subseteq V_t)$ and $N'(\subseteq V_t)$, the minimal materialized tree for N over t is a rooted subtree of the minimal materialized tree for N' over t if $N \subseteq N'$.*

Hence, t_p is a rooted subtree of t_S if $ENS(p, t) \subseteq ENS(S, t)$. However, there could be some patterns that t_S can totally answer but their embedding node sets are not included in $ENS(S, t)$. The following is an example.

EXAMPLE 4.8. *In Fig. 3, the exported XML tree t is shown in (b), and two patterns p_1 and p_2 are shown in (a) and (c) separately. We have that $ENS(p_1, t) = \{1, 2, 3, 8\}$*

and $ENS(p_2, t) = \{1, 2, 5, 8\}$. The XML trees t_{p_1} and t_{p_2} represented by p_1 and p_2 are shown in (d) and (f) respectively. Assume we cache t_{p_1} and want to answer p_2. Although $ENS(p_2, t) \not\subseteq ENS(p_1, t)$ due to the node with id = 5 in $ENS(p_2, t)$, t_{p_2} is a rooted subtree of t_{p_1}, i.e., t_{p_1} can totally answer p_2. The reason is that if a rooted subtree of t has the node with id = 8 from t, then it will also have node with id = 5 because the node with id = 5 is an ancestor of node with id = 8 in t. Hence, the node with id = 5 is redundant for $ENS(p_2, t)$ in representing an XML tree.

Furthermore, for a given pattern p, the nodes in $ENS(p, t)$ mapped from the internal nodes of p are redundant to represent an XML tree. The following definitions and lemmas are given to deal with this case.

DEFINITION 4.9. *Given an XML tree t and a tree pattern $p\langle V_p, E_p, r_p, o_p\rangle$, an **embedding leaf node set** $ELNS(p, t)$ is defined as $\cup_{e \in EB}(\cup_{v_i \in V_p^{leaf}}\{e(v_i).id\})$, where EB is a set including all possible embeddings from p to t and $V_p^{leaf} \subseteq V_p$ includes all leaf nodes of p.*

In above example, both embedding leaf node sets for patterns p_1 and p_2 over the exported XML tree t are $\{1, 2, 8\}$.

For an exported XML tree t, we similarly define that the embedding leaf node set for a pattern set S, denoted as $ELNS(S, t)$, is the union of the embedding leaf node set for each $p_i \in S$, i.e., $\cup_{p_i \in S} ELNS(p_i, t)$. We have the following results:

LEMMA 4.10. *Let t be an exported XML tree. For a given pattern p and a pattern set S, the following hold:*

- *The minimal materialized tree for $ENS(p, t)$ over t (i.e., t_p) is identical to the minimal materialized tree for $ELNS(p, t)$ over t.*
- *The minimal materialized tree for $ENS(S, t)$ over t (i.e., t_S) is identical to the minimal materialized tree for $ELNS(S, t)$ over t.*

Following Lemma 4.6, 4.7 and 4.10, we easily have that t_p is a rooted subtree of t_S if $ELNS(p, t) \subseteq ELNS(S, t)$.

So far, we reduce the problem of deciding whether t_S can totally answer p or not to the problem that whether $ELNS(p, t) \subseteq ELNS(S, t)$ or not. We next consider how to decide $ELNS(p, t) \subseteq ELNS(S, t)$.

We denote a pattern $p\langle V_p, E_p, r_p, o_p\rangle$ as p_{o_p}, where $o_p \in V_p$ is the output node. We also can choose any node in V_p as the output node of p. For example, the pattern p with a node v_1 instead of o_p as the output node can be denoted as p_{v_1}. For a pattern p, we introduce a tree pattern set(**TPS**) including all patterns by choosing each node in p as the output node, and this pattern set can be formally defined as $\cup_{v_i \in V_p}\{p_{v_i}\}$ and denoted as TPS_p. If we only choose leaf nodes in p as the output node to build the set, we denote it as TPS_p^{leaf}.

EXAMPLE 4.11. *In Fig. 3, the pattern p_1 shown in (a) has one leaf node with label 'b' as its output node. This pattern has another leaf node with label 'c'. If we choose it as output node, we can have another pattern p_3 shown in (e). Hence, the pattern set $TPS_{p_1}^{leaf}$ for p_1 is $\{p_1, p_3\}$.*

Our next result shows that the node set $ELNS(p, t)$ is equal to the result of evaluating the pattern set TPS_p^{leaf} over t.

LEMMA 4.12. *Given an XML tree t and a pattern p, $ELNS(p, t) = TPS_p^{leaf}(t)$*

For a pattern set S, we similarly define a pattern set TPS_S^{leaf} as $\cup_{p_i \in S} TPS_{p_i}^{leaf}$. From the above lemma, we can easily have $ELNS(S, t) = TPS_S^{leaf}(t)$.

By combining all above lemmas, we finally have the following conclusion:

THEOREM 2. *Given an exported XML tree t, a pattern p and a pattern set S, t_S can totally answer p if $TPS_p^{leaf} \sqsubseteq TPS_S^{leaf}$.*

Hence, we reduce the problem deciding whether t_S can totally answer p or not to the containment problem between two pattern sets TPS_p^{leaf} and TPS_S^{leaf}. We will discuss the complexity of this problem and corresponding algorithms in Section 5.

4.3 Incremental Maintenance of the Cached XML Tree

When the XML data represented by some patterns in our caching system are expired or less queried by clients, we need to consider clearing out them and incorporating new data. In this subsection, we discuss how to incrementally maintain the cached XML tree when a pattern is added to or removed from the semantic scheme, which is a pattern set.

Given an XML tree t exported by the server, let's assume we already have t_S for a pattern set S. When we want to add a pattern p to S, our problem is how to get $t_{S \cup \{p\}}$ from t_S. The idea is that we acquire t_p from server first, and then **merge** t_p to t_S.

We can get t_p by pruning all nodes in t whose descendants don't include any node in $ELNS(p, t)$. The **merge** of t_p and t_S works as follows: We assume that t_S has nodes $e_1, ..., e_k$ as its children, and t_p has nodes $n_1, ..., n_l$ as its children. For each node $e_i (1 \leq i \leq k)$, if there is a node $n_j (1 \leq j \leq l)$ that has the same id as e_i, we recursively merge the subtree of t_p, which is rooted at n_j and includes all its descendants, to the subtree of t_S, which is rooted at e_i and all its descendants; otherwise, we put the subtree rooted at n_j under the root t_S as its new subtree. We have the following result about this merged tree:

LEMMA 4.13. *Given a pattern p and a pattern set S, the tree merged from t_p and t_S is the minimal materialized tree for $ELNS(S \cup \{p\}, t)$, i.e., $t_{S \cup \{p\}}$.*

When we remove a pattern p from S, our problem is how to get $t_{S\setminus\{p\}}$. We only need to compute $ELNS(S\setminus\{p\},t)$. Similar to acquire t_p, we can get $t_{S\setminus\{p\}}$ by pruning all nodes in t_S whose descendants don't include any node in $ELNS(S\setminus\{p\},t)$.

5. COMPLEXITY

Obviously, incrementally maintaining the cached XML tree can be solved polynomially. In this section, we mainly discuss the complexity of deciding containment between two pattern sets.

The complexity of the pattern (not pattern set) containment problem has been well studied [13] and also for its three subclasses, which only use two of the three features: '//', '[]'and '*'in addition to '/'. The problem is in co-NP complete [13] for $XP^{\{/,//,[],*\}}$ and in P for its three subclasses [3, 14, 15].

[13] showed that the containment problem between one pattern and one pattern set can be reduced to that between two patterns. Hence, the containment problem between two pattern sets is still in coNP-complete. Furthermore, in case that all patterns in two pattern sets don't include '//'or '*', the pattern set containment problem can be decided in polynomial time. However, when patterns have '//', '*' but no '[]', this problem between two pattern sets is in coNP-complete even though this problem between two patterns is in P.

The only heuristic polynomial-time algorithm to decide containment between two patterns is to find a homomorphism between them. This algorithm is practical and sound for patterns, and also complete for its three subclasses. Based on this algorithm, we can also give a heuristic algorithm to decide containment between two pattern sets: for two pattern sets S_1 and S_2, this algorithm reports $S_1 \sqsubseteq S_2$ if $\forall p_1^i \in S_1\ \exists p_2^j \in S_2$ s.t. $p_1^i \sqsubseteq p_2^j$. We also use finding homomorphisms algorithm to decide $p_1^i \sqsubseteq p_2^j$.

6. RELATED WORK

Semantic Caching has been studied a lot in the relational model [4, 10]. Recently, semantic caching has attracted moderate attentions in XML world [7, 16, 5, 1, 12]. In [7], Chen et al consider using cached results of previous XQuery queries to answer new queries. In [16], Yang et al consider mining frequent tree patterns to cache their results for answering new queries. In both works, only queries, whose results have already been cached, can be answered. In [5], Balmin et al consider using pre-computed results of XPath queries also with data values, full paths, or node references to speedup processing of XPath queries. However, this work rules out the combination of results of multiple XPath queries in evaluating XPath queries. The most similar work to ours is [1, 12]. They consider prefix-selection queries that can also be represented as a tree. However, those queries don't support the important feature of XPath language: descendant axes, and also have strong constraints on structures of query trees, for example, two siblings must have different labels in a query tree.

7. CONCLUSION

This paper has introduced a novel framework for a new semantic caching system. The proposed framework offers the representation system of cached XML data, the algorithms to decide whether a new query can be totally answer by cached XML data or not, and to incrementally maintain cached XML data.

8. REFERENCES

[1] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying xml with incomplete information. In *PODS*, 2001.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: a primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.

[3] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *ICDE*, pages 493–504, 2003.

[5] A. Balmin, F. Ozcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, pages 60–71, 2004.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language, November 2003.

[7] L. Chen and E. A. Rundensteiner. Ace-xq: A cache-aware xquery answering system. In *WebDB*, pages 31–36, 2002.

[8] Y. Chen, S. B. Davidson, and Y. Zheng. Blas: An efficient xpath processing system. In *SIGMOD*, pages 47–58, 2004.

[9] J. Clark. Xml path language (xpath).

[10] S. Dar, M. J. Franklin, and B. Jonsson. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[11] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.

[12] V. Hristidis and M. Petropoulos. Semantic caching of xml databases. In *WebDB*, pages 25–30, 2002.

[13] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS*, pages 65–76, 2002.

[14] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[15] P. T. Wood. Minimizing simple xpath expressions. In *WebDB*, pages 13–18, 2001.

[16] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of xml query patterns for caching. In *VLDB*, pages 69–80, 2003.